R for Non-Programmers: A Guide for Social Scientists

Daniel Dauber

2021-08-18

Contents

Welcome

Welcome to *R for Non-Programmers: A guide for Social Scientists*. This book is intended to be of help to everyone who wishes to enter the world of R Programming, but not necessarily for the purpose to become a programmer. Instead, this book intends to convey key concepts in data analysis, especially quantitative research.

[Add more]

Acknowledgments

Special thanks are due to my wife, who supported me in so many ways to get this book completed. Also, I would like to thank my son, who patiently watched me sitting at the computer type this book up.

Chapter 1

`Readme.` Before you get started

1.1 A starting point and reference book

`to be added`

1.2 Download the companion R package

`to be added`

- Explain where to download
- Explain how examples are shown in this book (code in the first grey box, console results in the second grey box

1.3 A 'tidyverse' approach with some basic R

`to be added`

Chapter 2

Why learn a programming language as a non-programmer?

'R', it is not just a letter you learn in primary school, but a powerful programming language. While it is used for a lot of quantitative data analysis, it has grown over the years to become a powerful tool that excels (#no-pun-intended) in handling data and performing customised computations with quantitative and qualitative data.

R is now one of my core tools to perform various types of analysis because I can use it in many different ways, for example,

- statistical analysis,
- corpus analysis,
- development of online dashboards to dynamically generate interactive data visualisations,
- connection to social media APIs for data collection,
- Creation of reporting systems to provide individualised feedback to research participants,
- Drafting and writing research articles, etc.

> *Learning R is like learning a foreign language. If you like learning languages, then 'R' is just another one.*

While *R* has become a comprehensive tool for data scientists, it has yet to find its way into the mainstream field of Social Sciences. Why? Well, learning programming languages is not necessarily something that feels comfortable to

everyone. It is not like Microsoft Word, where you can open the software and explore it through trial and error. Learning a programming language is like learning a foreign language: You have to learn vocabulary, grammar and syntax. Similar to learning a new language, programming languages also have steep learning curves and require quite some commitment.

For this reason, most people do not even dare to learn it because it is time-consuming and often not considered a *'core method'* in Social Sciences disciplines. Apart from that, tools like SPSS have very intuitive interfaces, which seem much easier to use (or not?). However, the feeling of having *'mastered' R* (although one might never be able to claim this) can be extremely rewarding.

I guess this introduction was not necessarily helpful in convincing you to learn any programming language. However, despite those initial hurdles, there are a series of advantages to consider. Below I list some good reasons to learn a programming language as they pertain to my own experiences.

2.1 Learning new tools to analyse your data is always essential

Theories change over time, and new insights into certain social phenomena are published every day. Thus, your knowledge might get outdated quite quickly. This is not so much the case for research methods knowledge. Typically, analytical techniques remain over many years. We still use the mean, mode, quartiles, standard deviation, etc., to describe our quantitative data. Still, there are always new computational methods that help us to crunch the numbers even more. *R* is a tool that allows you to venture into new analytical territory because it is open source. Thousands of developers provide cutting-edge research methods free of charge for you to try with your data. You can find them on platforms like GitHub. *R* is like a giant supermarket, where all products are available for free. However, to read the labels on the product packaging and understand what they are, you have to learn the language used in this supermarket.

2.2 Programming languages enhance your conceptual thinking

While I have no empirical evidence for this, I am very certain it is true. While I would argue that my conceptual thinking is quite good, I would not necessarily say that I was born with it. Programming languages are very logical. Any error in your code will make you fail to execute it properly. Sometimes you face challenges in creating the correct code to solve a problem. Through creative abstract thinking (I should copyright this term), you start to approach your problems differently, whether it is a coding problem or a problem in any other

context. For example, I know many students enjoy the process of qualitative coding. However, they often struggle to detach their insights from the actual data and synthesise ideas on an abstract and more generic level. Qualitative researchers might refer to this as challenges in '*second-order deconstruction of meaning'*. This process of abstraction is a skill that needs to be honed, nurtured and practised. From my experience, programming languages are one way to achieve this, but they might not be recognised for this just yet.

2.3 Programming languages allow you to look at your data from a different angle

There are certainly commonly known and well-established techniques regarding how you should analyse your data rigorously. However, it can be quite some fun to try techniques outside your disciplines. This does not only apply to programming languages, of course. Sometimes, learning about a new research method enables you to look at your current tools in very different ways too. One of the biggest challenges for any researcher is to reflect on your work. Learning new and maybe even *'strange'* tools can help with this. Admittedly, sometimes you might find out that some new tools are also a dead-end. Still, you might have learned something valuable through the process of engaging with your data differently. So shake off the rust of your analytical routine and blow some fresh air into your research methods.

2.4 Learning any programming language will help you learn other programming languages.

Once you understand the logic of one language, you will find it relatively easy to understand new programming languages. Of course, if you wanted to, you could become the next '*Neo'* (from 'The Matrix') and change the reality of your research forever. On a more serious note, though, if you know any programming language already, learning R will be easier because you have accrued some basic understanding of these particular types of languages.

Having considered everything of the above, do you feel ready for your next foreign language?

Chapter 3

Setting up R and RStudio

Every journey starts with gathering the right equipment. This intellectual journey is not much different. The first step that every 'R novice has to face is to set everything up to get started. There are essentially two strategies:

- Install *R*and RStudio

or

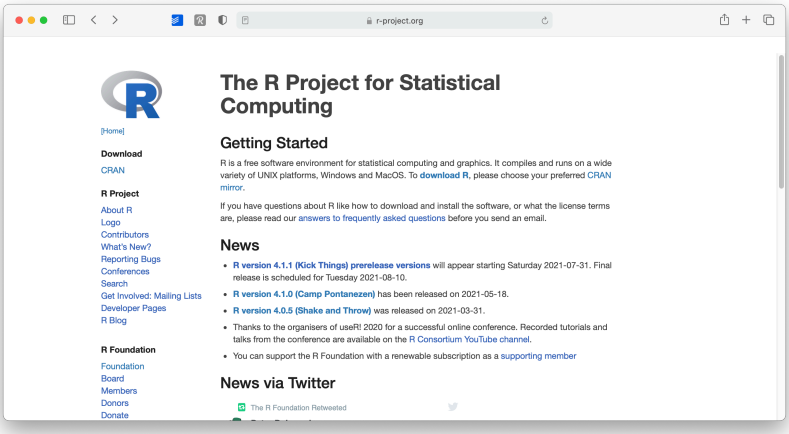
- Run RStudio in a browser via RStudio Cloud

While installing *R* and Studio requires more time and effort, I strongly recommend it, especially if you want to work offline or make good use of your computer's CPU. However, if you are not sure yet whether you enjoy learning *R*, you might wish to look at RStudio Cloud first. Either way, you can follow the examples of this book no matter which choice you make.

3.1 Installing R

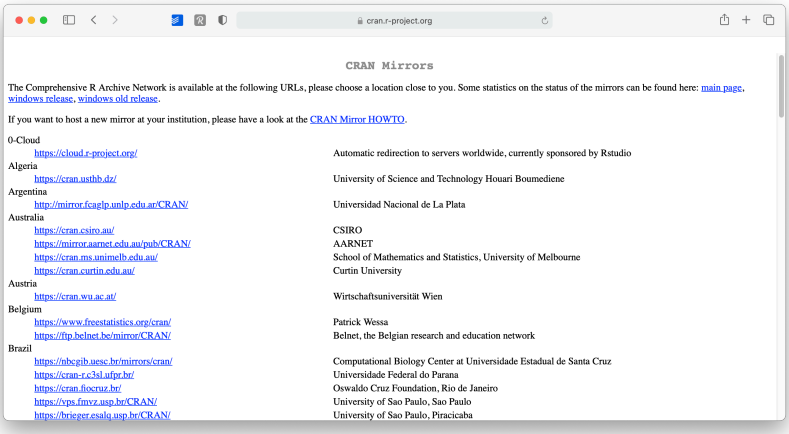
The core module of our programming is R itself, and since it is an open-source project, it is available for free on Windows, Mac and Linux computers. Here is what you need to do to install it properly on your computer of choice:

1. Go to www.r-project.org

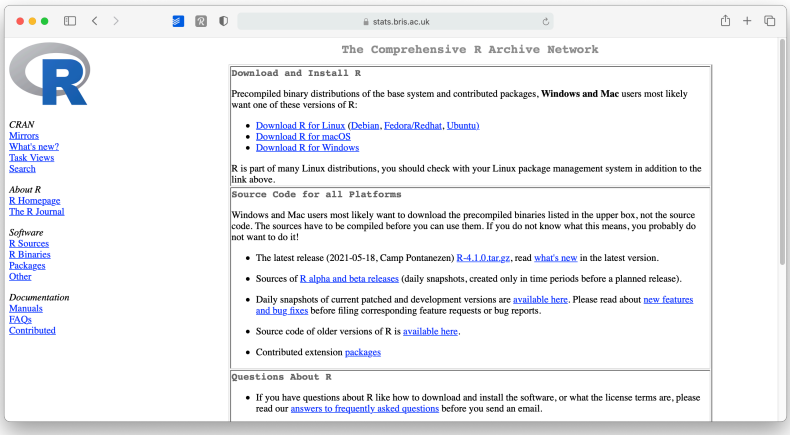


2. Click on `CRAN` where it says download.

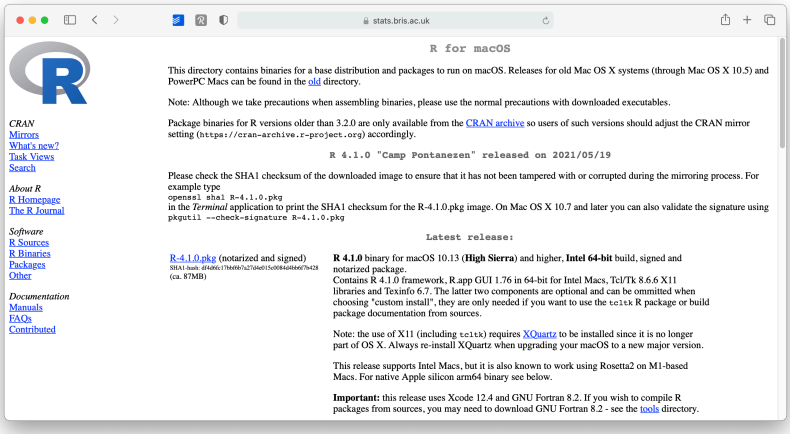
3. Choose a server in your country (all of them work, but downloads will perform quicker).



4. Select the operating system for your computer.



5. Select the version you want to install (I recommend the latest version)



6. Open the downloaded file and follow the installation instructions. (I recommend leaving the suggested settings as they are).

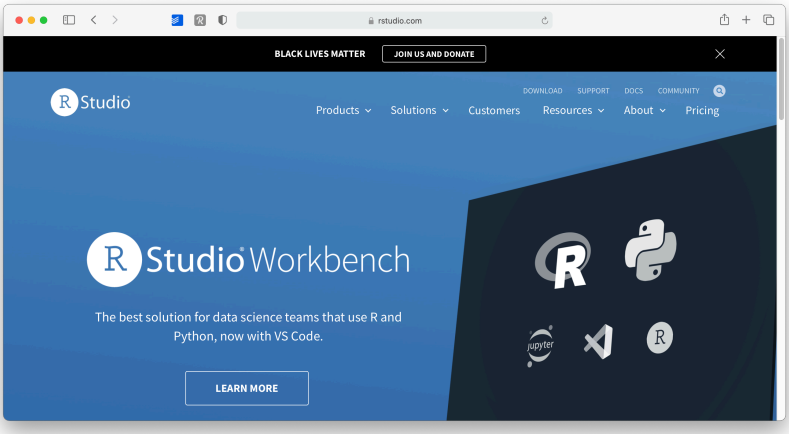
This was relatively easy. You now have *R* installed. Technically you can start using *R* for your research, but there is one more tool I strongly advise installing: RStudio.

3.2 Installing RStudio

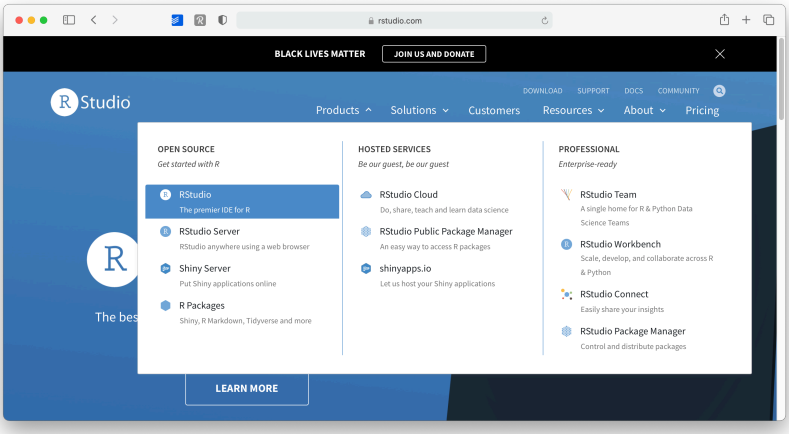
R by itself is just the *'beating heart'* of *R* programming, but it has no particular user interface. If you want buttons to click and actually 'see' what you

are doing, there is no better way than RStudio. RStudio is an *integrated development environment* (IDE) and will be our primary tool to interact with *R*. It is the only software you need to do all the fun parts and, of course, to follow along with the examples of this book. To install RStudio perform the following steps:

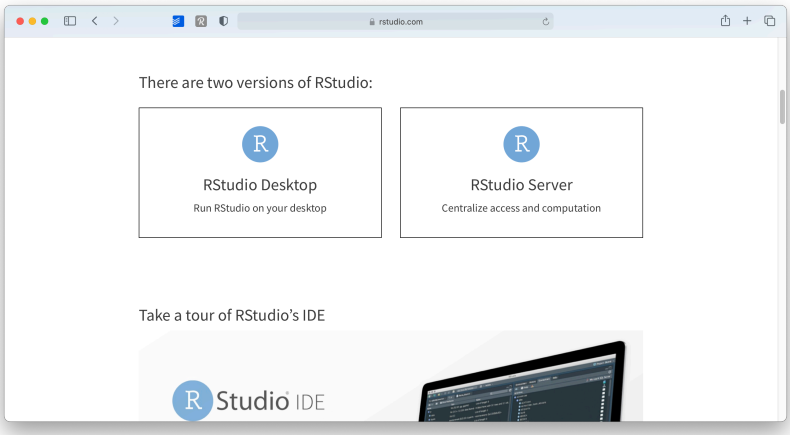
1. Go to www.rstudio.com.



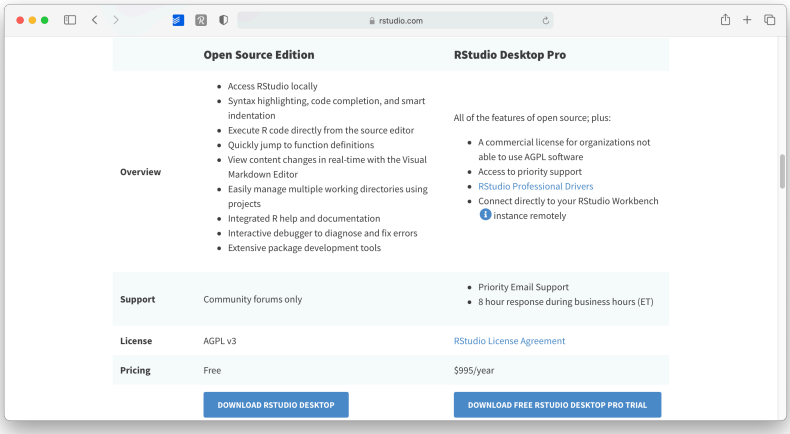
2. Go to `Products > RStudio`



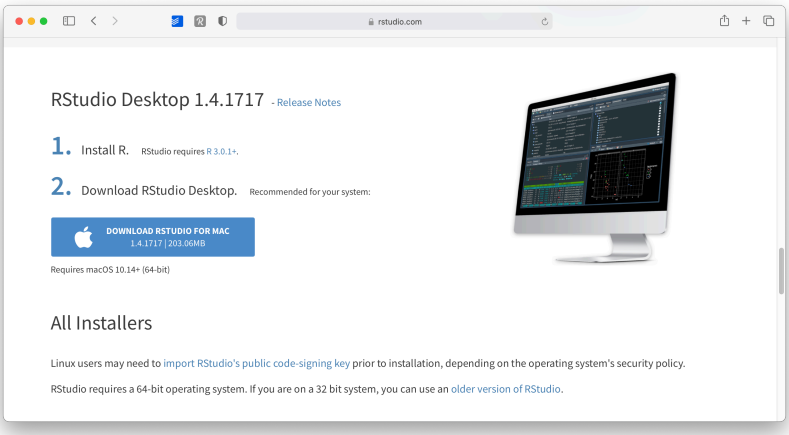
3. On this page, scroll down and select `RStudio Desktop`



4. Select the `'Open Source Edition'` option by clicking on `'Download RStudio Desktop'`



5. As a last step, scroll down where it shows you a download button for your operating system. The website will automatically detect this. You also get a nice reminder to install 'R' first, in case you have not done so yet.



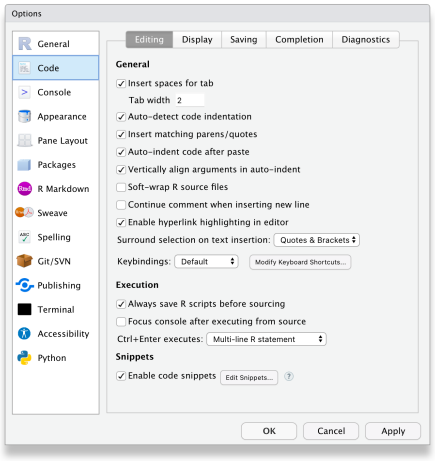
6. Open the downloaded file and follow the installation instructions (again, keep it to the default settings as much as possible)

Congratulations, you are all set up to learn *R*. From now on you only need to start RStudio and not *R*. Of course, if you are the curious, nothing shall stop you to try *R* without RStudio.

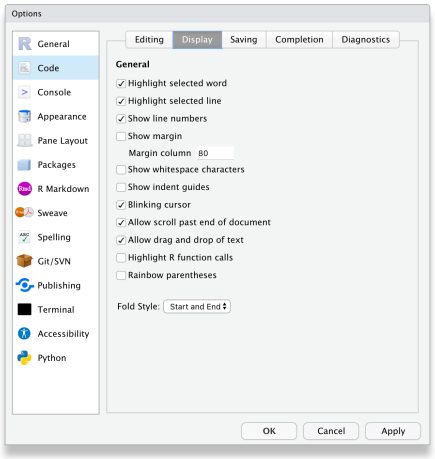
3.3 When you first start RStudio

Before you start programming away, you might want to make some tweaks to your settings right away to have a better experience (in my humble opinion). I recommend at least the following two changes by clicking on `RStudio > Preferences` or press `/Ctrl + ,`.

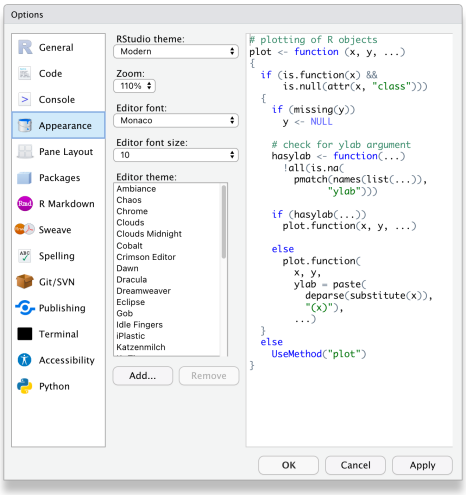
1. In the `Code > Editing` tab, make sure to have at least the first five options ticked, especially the `Auto-indent code after paste`. This setting will save time when trying to format your coding appropriately, making it easier to read. Indentation is the primary way of making your code look more readable and less like a series of characters that appear almost random.



2. In the `Display` tab, you might want to have the first three options selected. In particular, `Highlight selected line` is helpful because, in more complicated code, it is helpful to see where your cursor is.



Of course, if you wish to customise your workspace further, you can do so. The visually most impactful way to alter the default appearance of RStudio is to select Appearance and pick a completely different colour theme. Feel free to browse through various options and see what you prefer. There is no right or wrong here.



3.4 Updating R and RStudio: Living at the pulse of innovation

While not strictly something that helps you become a better programmer, this advice might come in handy to avoid turning into a frustrated programmer. When you update your software, you need to update R and RStudio separately from each other. While both R and RStudio work closely with each other, they still constitute separate pieces of software. Thus, it is essential to keep in mind that updating RStudio will not automatically update R. This can become problematic if specific packages you installed via RStudio (like a fancy learning algorithm) might not be compatible with earlier versions of R. Also, additional R packages developed by other people are separate pieces and are updated too, independently from R and RStudio.

I know what you are thinking: This already sounds complicated and cumbersome. However, rest assured, we take a look at how you can easily update all your packages with RStudio. Thus, all you need to remember is: *R* needs to be updated separately from everything else.

3.5 RStudio Cloud

`to be completed`

Chapter 4

The RStudio Interface

The RStudio interface is composed of quadrants, each of which fulfils a unique purpose:

- The `Console` window,
- The `Source` window,
- The `Environment / History / Connections / Tutorial` window, and
- The `Files / Plots / Packages / Help / Viewer` window

You might only see three windows and wonder where the `Source` window has gone in your version of RStudio. In order to use it you have to either open a file or create a new one. You can create a new file by selecting `File > New File > R Script` in the menu bar, or use the keyboard shortcut `Ctrl+Shift+N` on PC and `Cmd+Shift+N` on Mac.

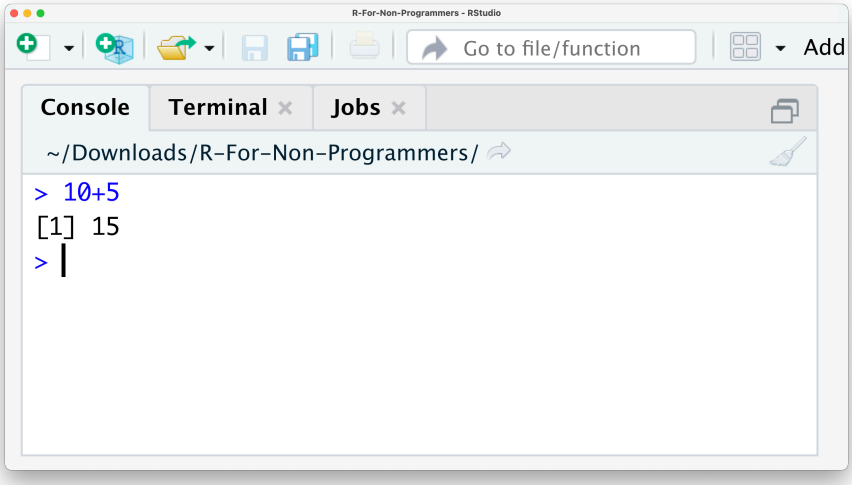
I will briefly explain the purpose of each window/pane and how they are relevant to your work in *R*.

4.1 The Console window

The console is located in the bottom-left, and it is where you often will find the output of your coding and computations. It is also possible to write code directly into the console. Let's try the following example by calculating the sum of `10 + 5`. Click into the console with your mouse, type the calculation into your console and hit `Enter/Return` on your keyboard. The result should be pretty obvious:

```
# We type the below into the console
10+5
## [1] 15
```

Here is a screenshot of how it should look like at your end in RStudio:



You just successfully performed your first successful computation. I know, this is not quite impressive just yet. *R* is undoubtedly more than just a giant calculator.

In the top right of the console, you find a symbol that looks like a broom. This one is quite an important one because it clears your console. Sometimes the console can become very cluttered and difficult to read. If you want to remove whatever you computed, you can click the broom icon and clear the console of all text. I use it so frequently that I strongly recommend learning the keyboard shortcut, which is `Ctrl+L` on PC and Mac.

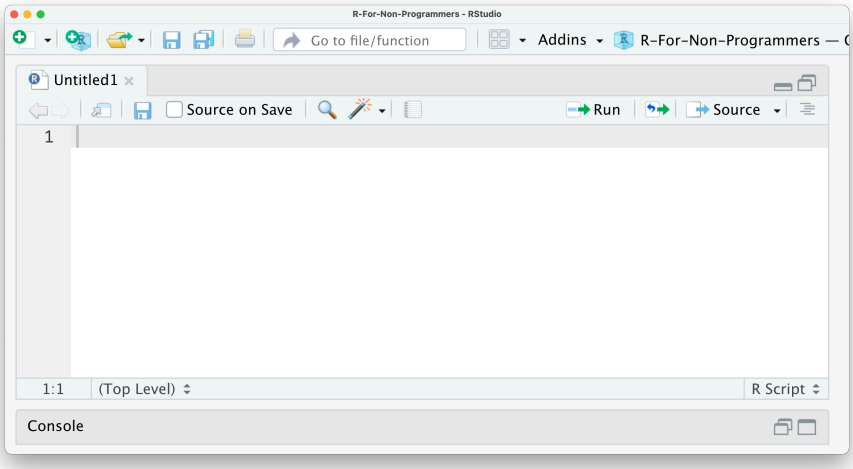
4.2 The Source window

In the top left, you can find the source window. The term 'source' can be understood as any type of file, e.g. data, programming code, notes, etc. The source panel can fulfil many functions, such as:

- Inspect data in an Excel-like format (LINK TO RELEVANT CHAPTER)
- Open programming code, e.g. an R Script (LINK TO RELEVANT CHAPTER)
- Open other text-based file formats, e.g.

  - Plain text (.txt),
  - Markdown (.md),
  - Websites (.html),
  - LaTeX (.tex),
  - BibTex (.bib),

- Edit scripts with code in it,
- Run the analysis you have written.

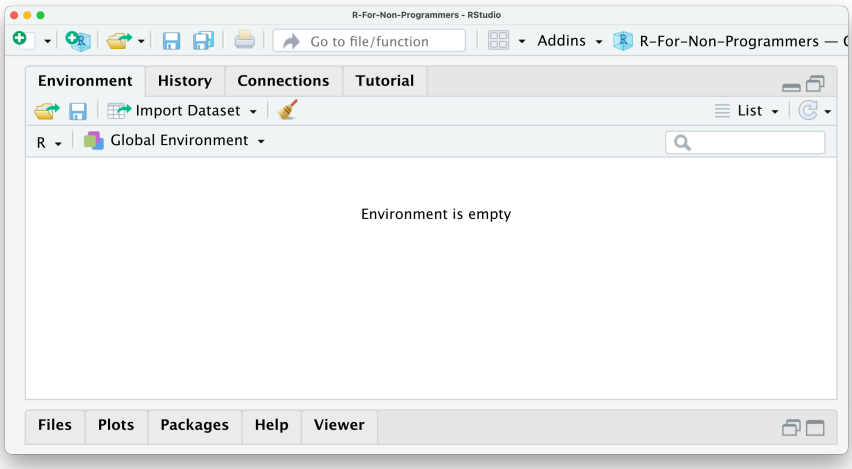


In other words, the source window will show you whatever file you are interested in, as long as RStudio can read it - and no, Microsoft Office Documents are not supported. Another limitation of the source window is that it can only show text-based files. So opening images, etc. would not work.

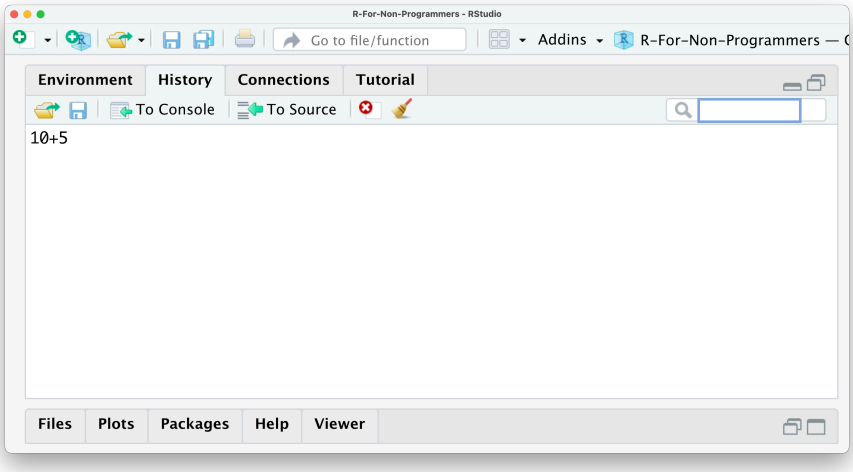
4.3 The Environment / History / Connections / Tutorial window

The window in the top right shows multiples panes. The first pane is called *Environment* and shows you objects which are available for computation. One of the first objects you will create is your dataset because, without data, we cannot perform any analysis. Thus, one object might be your data. Another object could be a plot showing the number of male and female participants in your study. To find out how to create objects yourself, you can take a glimpse at (INSERT CHAPTER X). Besides datasets and plots, you will also find other objects here, e.g. lists, vectors and functions you created yourself. Don't worry

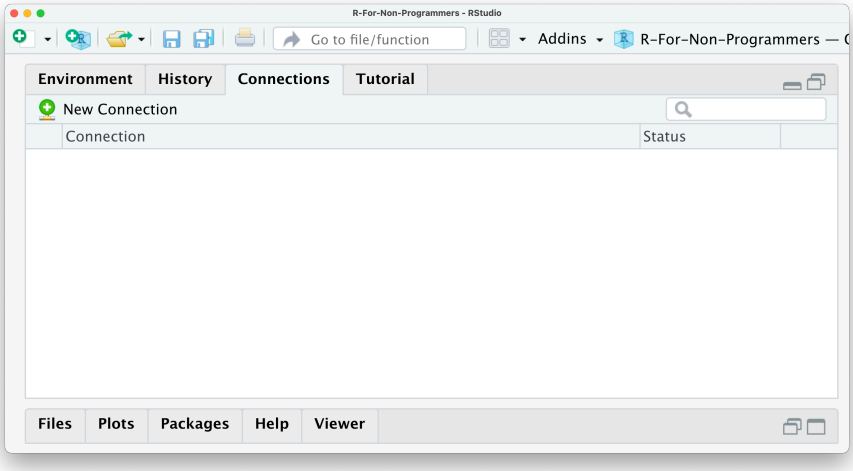
if none of these words makes sense at this point. We will cover each of them in the upcoming chapters. For now, remember this is a place where you can find different objects you created.



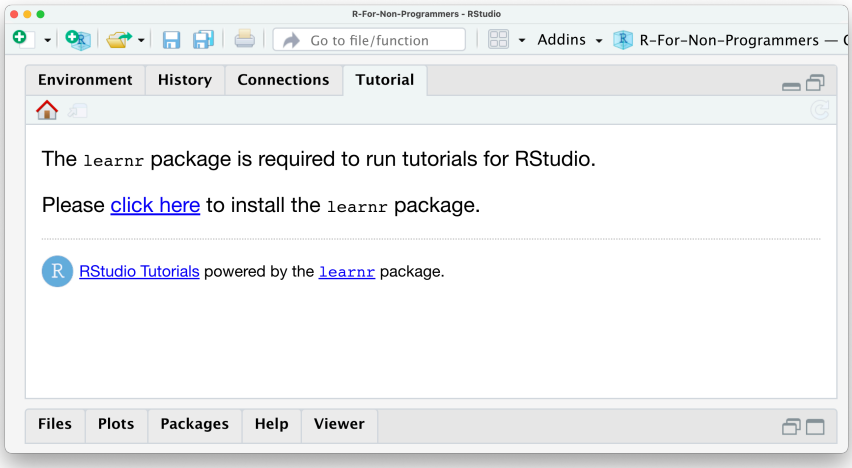
The *History* pane is very easy to understand. Whatever computation you run in the Console will be stored. So you can go back and see what you coded and rerun that code. Remember the example from above where we computed the sum of `10+5`? This computation is stored in the history of RStudio, and you can rerun it by clicking on `10+5` in the history pane and then click on `To Console`. This will insert `10+5` back into the Console, and we can hit `Return` to retrieve the result. You also have the option to copy the code into an existing or new R Script by clicking on `To Source`. By doing this, you can save this computation on your computer and reuse it later. Finally, if you would like to store your history, you can do so by clicking on the `floppy disk symbol`. There are two more buttons in this pane, one allows you to delete individual entries in the history, and the last one, a `broom`, clears the entire history (irrevocably).



The pane *Connections* allows you to tab into external databases directly. This can come in handy when you work collaboratively on the same data or want to work with extensive datasets without having to download them. However, for an introduction to R, we will not use this feature of RStudio for now.

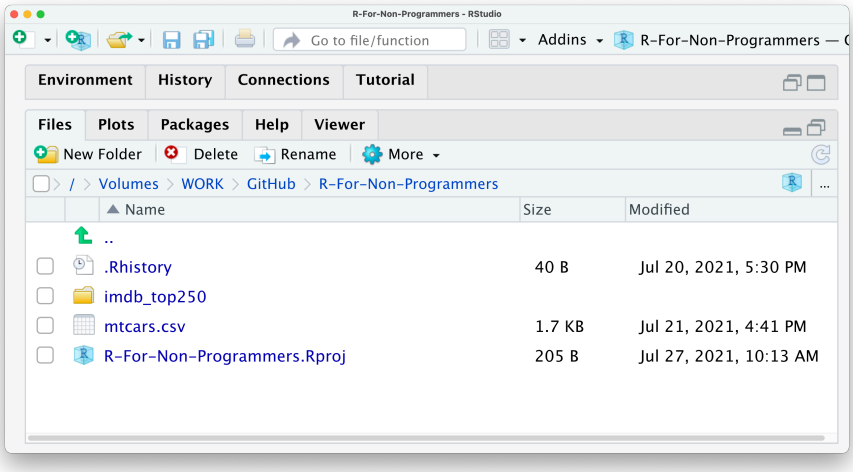


The last pane is called *Tutorial*. Here you can find additional materials to learn *R* and RStudio. If you search for more great content to learn R, this serves as a great starting point.



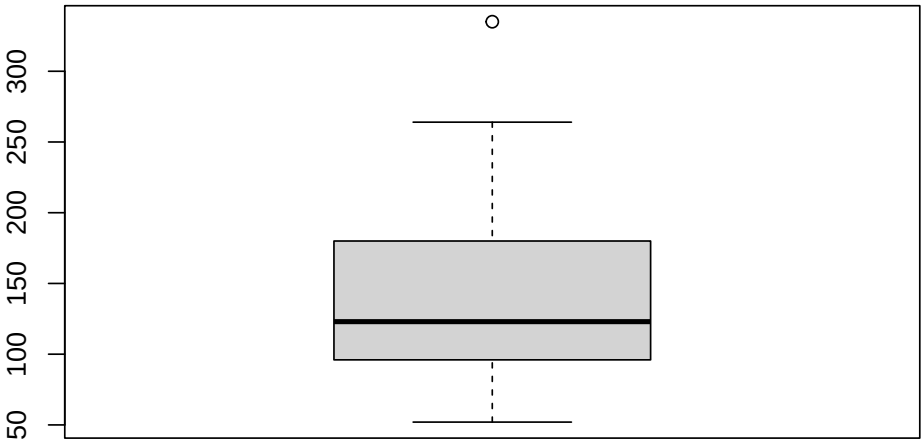
4.4 The Files / Plots / Packages / Help / Viewer window

The last window consists of five essential panes. The first one is the *Files* pane. As the name indicates, it lists all the files and folders in your root directory. A root directory is the default directory where RStudio saves your files, for example, your analysis. However, you can easily change this directory to something else (see also CHAPTER X) or use R Project files (see CHAPTER X) to carry out your research. Thus, the *Files* pane is an easy way to load data into RStudio and create folders to keep your research project well organised.



Since the Console cannot reproduce data visualisations, RStudio offers a way to do this very easily. It is through the Plots pane. This pane is exclusively designed to show you any plots you have created using R. Here is a simple example that you can try. Type into your console `boxplot(mtcars$hp)`.

```
# Here we create a nice boxplot using a dataset called 'mtcars'
boxplot(mtcars$hp)
```



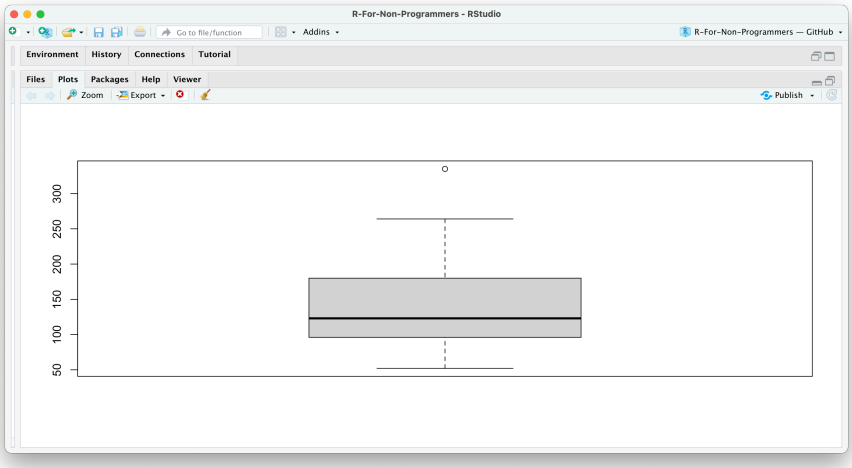
Although this is a short piece of coding, it performs quite a lot of steps:

- it uses a function called `boxplot()` to draw a boxplot of
- a variable called `hp` (for horsepower), which is located in

- a dataset named `mtcars`,

- and it renders the graph in your *Plots* pane

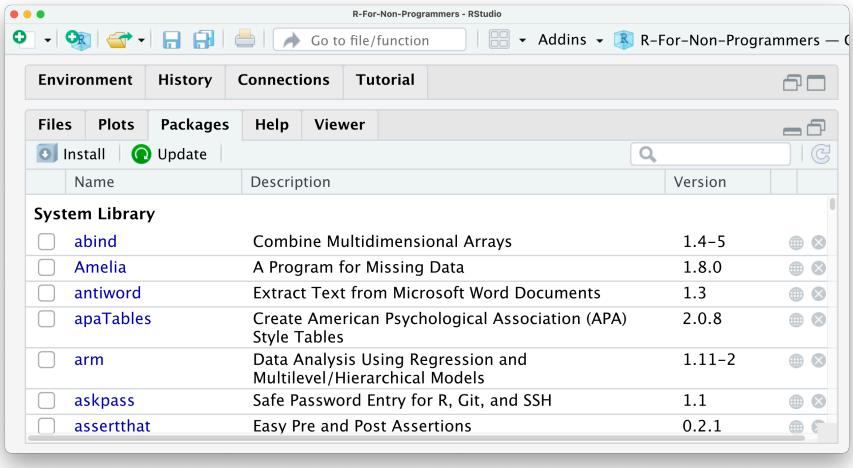
This is how the plot should look like in your RStudio *Plots* pane.



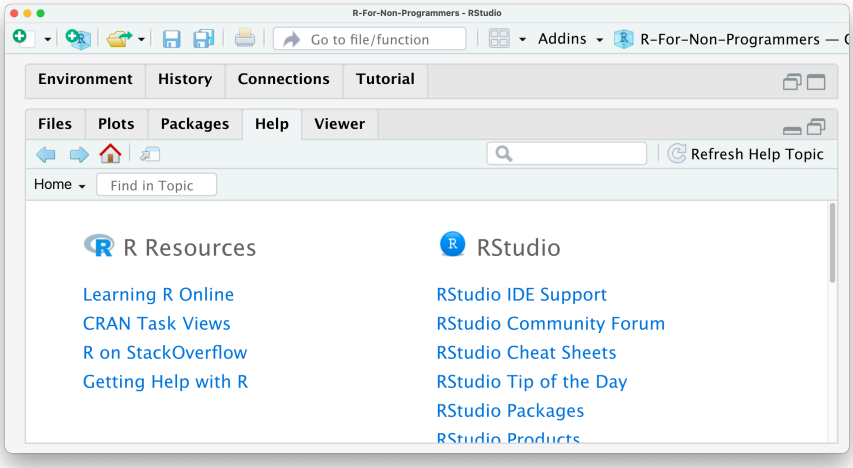
If you wish to delete the plot, you can click on the `red circle with a white x` symbol. This will delete the currently visible plot. If you wish to remove all plots from this pane, you can use the `broom`. There is also an option to export your plot and move back and forth between different plots.

Do not worry about the coding at this point. It will all make sense in the following chapters.

The next pane is called *Packages*. Packages are additional tools you can import and use when performing your analysis. A frequent analogy people use to explain packages is your phone and the apps you install. Each package you download is equivalent to an app on your phone. It can enhance different aspects of working in *R*, such as creating animated plots, using unique machine learning algorithms, or simply making your life easier by doing multiple computations with just one single line of code. You will learn more about *R packages* in Chapter 5.4.



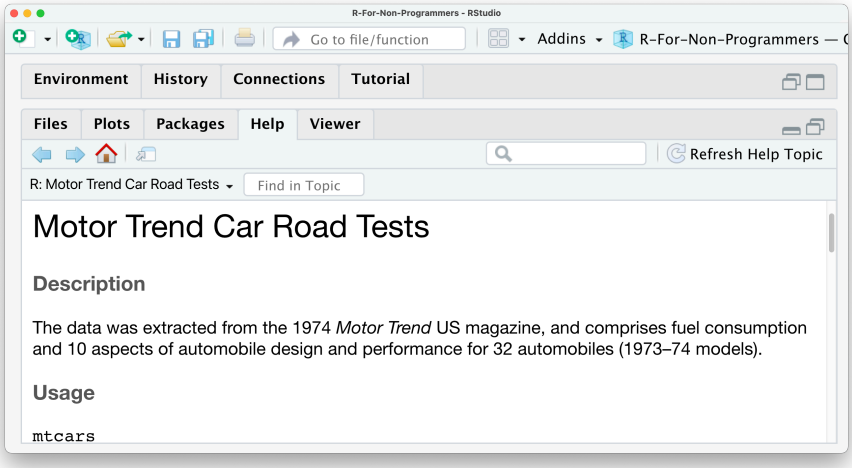
If you are in dire need of help, RStudio provides you with a *Help* pane. You can search for specific topics, for example how certain computations work. The *Help* pane also has documentation on different datasets that are included in *R*, RStudio or *R packages* you have installed. If you want a more comprehensive overview of how you can find help, have a look at CRAN's 'Getting Help with R' webpage.



So, for example, if you want to know what the `mtcars` dataset is, you can either use the search window in the *Help* pane or, much easier, use a `?` in the console to search for it:

```
# Type a '?' and immediately add the name to bring up helpful information.

?mtcars
```

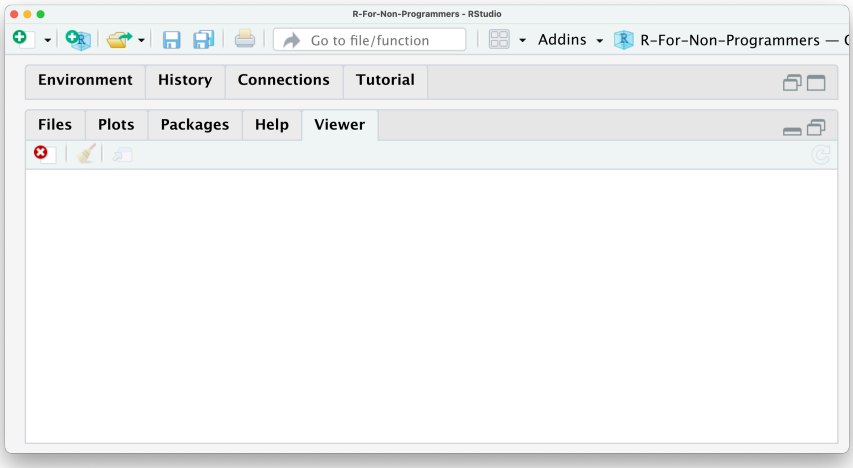
This will open the *Help* pane and give you more information about this dataset:



There are many different ways of how you can find help with your coding beyond RStudio and this book. My top three platforms to find solutions to my programming problems are:

- Google
- stackoverflow.com
- Twitter (with #RStats)

Lastly, we have the *Viewer* pane. Not every data visualisation we create in R is a static image. You can create dynamic data visualisations or even websites with R. This type of content is displayed in the Viewer pane rather than in the Plots pane. Often these visualisations are based on HTML and other web-based programming languages. As such, it is easy to open them in your browser as well. However, in this book, we mainly focus on two-dimensional static plots, which are the ones you likely need most of the time, either for your assignments, thesis, or publication.



4.5 Customise your user interface

As a last remark in this chapter, I would like to make you aware that you can modify each window. There are three basic adjustments you can make:

- Hide panes by clicking on the window symbol in the top right corner of each window,
- Resize panes by dragging the border of a window horizontally or vertically, or
- Add and remove panes by going to `RStudio > Preferences > Pane Layout`, or use the keyboard shortcut `+ ,` if you are on a Mac. There is, unfortunately no default shortcut for PC users.

If you want a fully customised experience you can also alter the colour scheme of the RStudio itself (`RStudio > Preferences > Appearance`) and if the themes offered are not enough for you, you can create a custom theme here

Chapter 5

R Basics: The very fundamentals

After a likely tedious installation of R and RStudio, as well as a somewhat detailed introduction to the RStudio interface, you are finally ready to 'do' things. By 'doing', I mean coding. The term 'coding' in itself can instil fear in some of you, but you only need one skill to do it: Writing. As mentioned earlier, learning coding or programming means learning a new language. However, once you have the basic grammar down, you already can communicate quite a bit. In this section, we will explore the fundamentals of R. These build the foundation for everything that follows. After that, we dive right into some analysis.

5.1 Basic computations in R

The most basic computation you can do in R is arithmetic operations. In other words, addition, subtraction, multiplication, division, exponentiation and extraction of roots. In other words, R can be used like your pocket calculator, or more likely the one you have on your phone. For example, in Chapter 4.1 we already performed an addition. Thus, it might not come as a surprise how their equivalents work in R. Let's take a look at the following examples:

```
# Addition
10 + 5
## [1] 15

# Subtraction
10 - 5
## [1] 5
```

```
# Multiplication
10 * 5
## [1] 50

# Division
10 / 5
## [1] 2

# Exponentiation
10 ^ 2
## [1] 100

# Square root
sqrt(10)
## [1] 3.162278
```

They all look fairly straightforward except for the extraction of roots. As you probably know, extracting the root would typically mean we use the symbol $\sqrt{}$ on your calculator. To compute the square root in R, we have to use a function instead to perform the computation. So we first put the name of the function `sqrt` and then the value `10` within parenthesis `()`. This results in the following code: `sqrt(10)`. If we were to write this down in our report, we would write $\sqrt[2]{10}$.

Functions are an essential part of R and programming in general. You will learn more about them in this chapter.Besides arithmetic operations, there are also logical queries you can perform. Logical queries always return either the value TRUE or FALSE. Here are some examples which make this clearer:

```
#1 Is it TRUE or FALSE?
1 == 1
## [1] TRUE

#2 Is 45 bigger than 55?
45 > 55
## [1] FALSE

#3 Is 1982 bigger or equal to 1982?
1982 >= 1982
## [1] TRUE

#4 Are these two words NOT the same?
"Friends" != "friends"
## [1] TRUE
```

```
#5 Are these sentences the same?
"I love statistics" == "I love statistícs"
## [1] FALSE
```

Reflecting on these examples, you might notice three important things:

1. I used `==` instead of `=`,
2. I can compare non-numerical values, i.e. text, which is also known as `character` values, with each other,
3. The devil is in the details (considering #5).

One of the most common mistakes of R novices is the confusion around the `==` and `=` notation. While `==` represents `equal to`, `=` is used to assign a value to an object (for more details on assignments see Chapter 4.4). However, in practice, most R programmers tend to avoid `=` since it can easily lead to confusion with `==`. As such, you can strike this one out of your R vocabulary for now.

There are many different logical operations you can perform. Table 5.1 lists the most frequently used logical operators for your reference. These will become important once we select only certain parts of our data for analysis, e.g. only `female` participants.

Table 5.1: Logical Operators in R

Operator	Description
==	is equal to
>=	is bigger or equal to
<=	is smaller of equal to
!=	is not equal to
a | b	a or b
a & b	a and b
!a	is not a

5.2 Assigning values to objects: '<-'

Another common task you will perform is assigning values to an object. An object can be many different things:

- a dataset,
- the results of a computation,
- a plot,

- a series of numbers,
- a list of names,
- a function,
- etc.

In short, an object is an umbrella term for many different things which form part of your data analysis. For example, objects are handy when storing results that you want to process further in later analytical steps. Let's have a look at an example.

```
# I have a friend called "Fiona"
friends <- "Fiona"
```

In this example, I created an object called `friends` and added `"Fiona"` to it. Remember, because `"Fiona"` represents a `string`, we need `""`. So, if you wanted to read this line of code, you would say, '`friends` gets the value `"Fiona"`'. Alternatively, you could also say '`"Fiona"` is assigned to `friends`'.

If you look into your environment pane, you will find the object we just created. You can see it carries the value `"Fiona"`. We can also print values of an object in the console by simply typing the name of the object `friends` and hit `Return` .

```
# Who are my friends?
friends
## [1] "Fiona"
```

Sadly, it seems I only have one friend. Luckily we can add some more, not the least to make me feel less lonely. To create objects with multiple values, we can use the function `c()`, which stands for 'concatenate'. The Cambridge Dictionary (2021) define this word as follows:

> '***concatenate***',
>
> to put things together as a connected series

Let's concatenate some more friends into our `friends` object.

```
# Adding some more friends to my life
friends <- c("Fiona", "Ida", "Lukas", "Georg", "Daniel", "Pavel", "Tigger")

# Here are all my friends
friends
## [1] "Fiona"  "Ida"    "Lukas"  "Georg"  "Daniel" "Pavel"  "Tigger"
```

To concatenate values into a single object, we need to use a comma `,` to separate each value. Otherwise, R will report an error back.

```
friends <- c("Fiona" "Ida")
## Error: <text>:1:22: unexpected string constant
## 1: friends <- c("Fiona" "Ida"
##                          ^
```

R's error messages tend to be very useful and give meaningful clues to what went wrong. In this case, we can see that something 'unexpected' happen, and it shows where our mistake is.

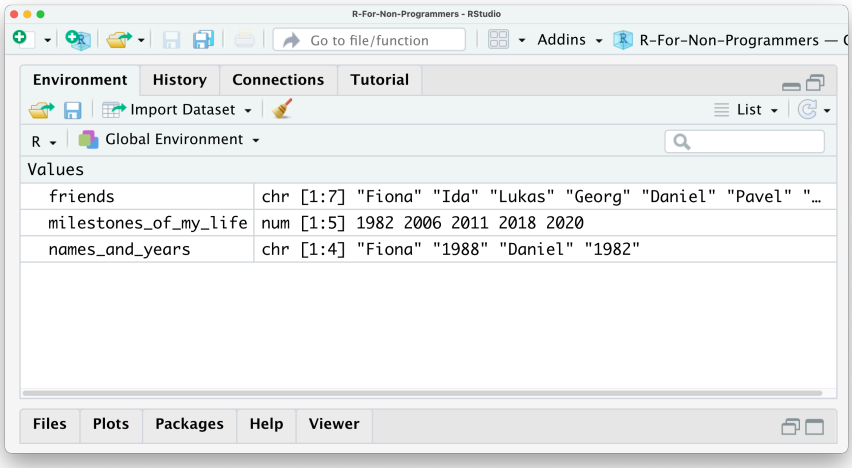
You can also concatenate numbers, and if you add `()` around it, you can automatically print the content of the object to the console. Thus, `(milestones_of_my_life <- c(1982, 2006, 2011, 2018, 2020))` is the same as `milestones_of_my_life <- c(1982, 2006, 2011, 2018, 2020)` followed by `milestones_of_my_life`. The following examples illustrate this.

```
# Important years in my life
milestones_of_my_life <- c(1982, 2006, 2011, 2018, 2020)
milestones_of_my_life
## [1] 1982 2006 2011 2018 2020

# The same as above, but we don't need the second line of code
(milestones_of_my_life <- c(1982, 2006, 2011, 2018, 2020))
## [1] 1982 2006 2011 2018 2020
```

Finally, we can also concatenate numbers and character values into one object:

```
(names_and_years <- c("Fiona", 1988, "Daniel", 1982))
## [1] "Fiona"  "1988"   "Daniel" "1982"
```

This last example is not necessarily something I would recommend to do, because it likely leads to undesirable outcomes. If you look into your environment pane you currently have three objects: `friends`, `milestones_of_my_life`, and `names_and_years`.



The `friends` object shows that all the values inside the object are classified as `chr`, which denominates `character`. In this case, this is correct because it only includes the names of my friends. On the other hand, the object `milestones_of_my_life` only includes `numeric` values, and therefore it says `num` in the environment pane. However, for the object `names_and_years` we know we want to have `numeric` and `character` values included. Still, R recognises them as `character` values only because values inside objects are meant to be of the same type.

Consequently, mixing different types of data (as explained in Chapter @ref()) into one object is likely a bad idea. This is especially true if you want to use the numeric values for computation. In short: ensure your objects are all of the same data type.

There is an exception to this rule. 'Of course', you might say. There is one object that can have values of different types: `list`. As the name indicates, a `list` object holds several items. These items are usually other objects. In the spirit of 'Inception', you can have lists inside lists, which contain more objects.

Let's create a list called `x_files` using the `list` function and place all our objects inside.

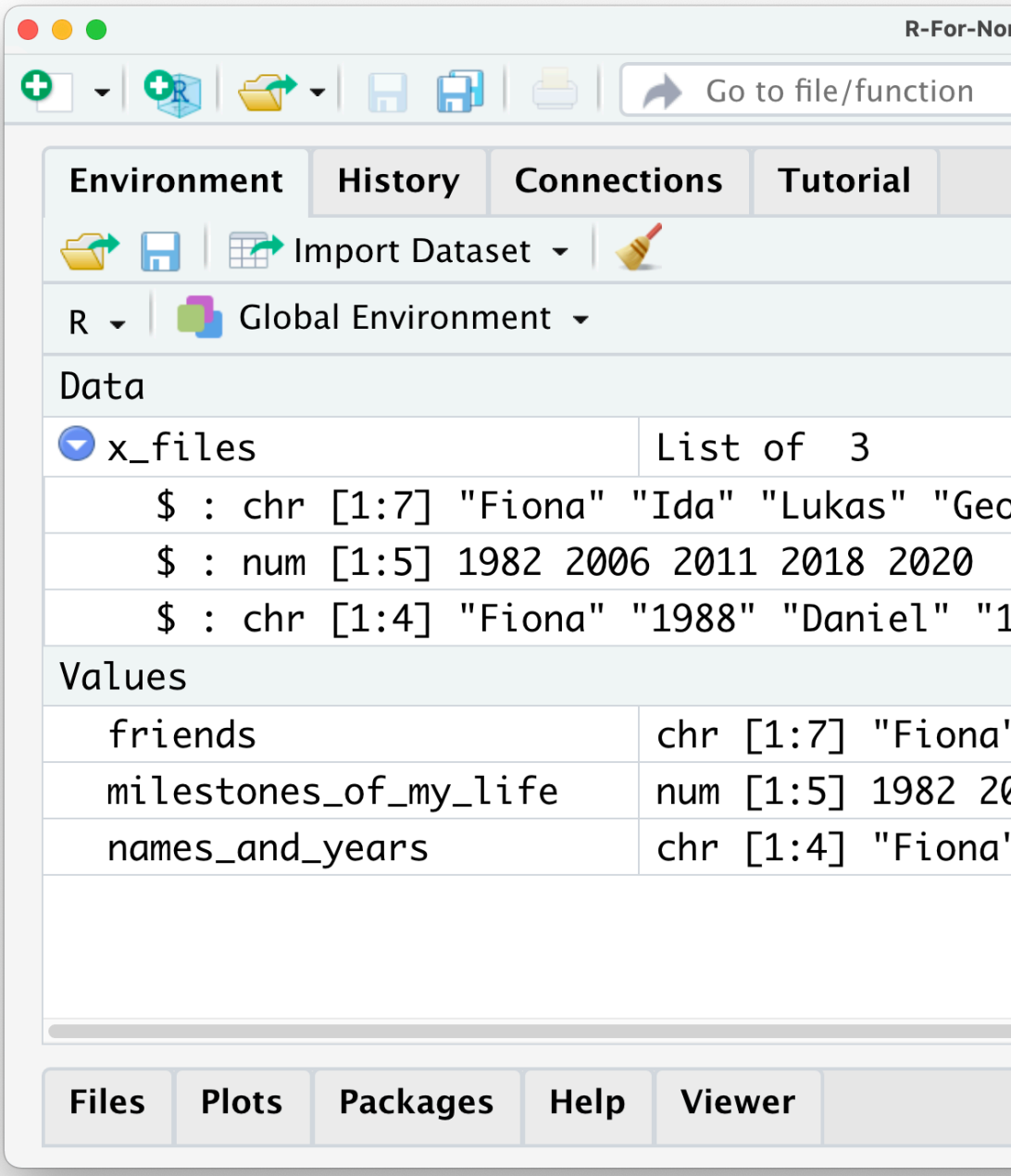
```
# This creates our list of objects
x_files <- list(friends,
                milestones_of_my_life,
                names_and_years)

# Let's have a look what is hidden inside the x_files
x_files
## [[1]]
```

```
## [1] "Fiona"  "Ida"    "Lukas"  "Georg"  "Daniel" "Pavel"  "Tigger"
##
## [[2]]
## [1] 1982 2006 2011 2018 2020
##
## [[3]]
## [1] "Fiona"  "1988"   "Daniel" "1982"
```

You will notice in this example that I do not use `""` for each value in the list. This is because `friends` is not a character I put into the list, but an object. When we refer to objects, we do not need quotation marks.

We will encounter `list` objects quite frequently when we perform our analysis. Some functions return the results in the format of lists. This can be very helpful because otherwise our environment pane will be littered with objects. We would not necessarily know how they relate to each other, or worse, to which analysis they belong. Looking at the list item in the environment page (Figure 5.2), you can see that the object `x_files` is classified as a `List of 3,` and if you click on the blue icon, you can inspect the different objects inside.

\begin{figure}


R-For-No
Go to file/function
Environment
History
Connections
Tutorial
Import Dataset
R
Global Environment
Data
x_files
List of 3
$: chr [1:7] "Fiona" "Ida" "Lukas" "Geo
$: num [1:5] 1982 2006 2011 2018 2020
$: chr [1:4] "Fiona" "1988" "Daniel" "1
Values
friends
chr [1:7] "Fiona'
milestones_of_my_life
num [1:5] 1982 20
names_and_years
chr [1:4] "Fiona'
Files
Plots
Packages
Help
Viewer

{

}

\caption{The environment pane showing our objects and our list `x_files`}
\end{figure}

In Chapter 5.1, I mentioned that we should avoid using the `=` operator and explained that it is used to assign values to objects. You can, if you want, use `=` instead of `<-`. They fulfil the same purpose. However, as mentioned before, it is not wise to do so. Here is an example that shows that, in principle, it is possible.

```
# DO
(avengers1 <- c("Iron Man", "Captain America", "Black Widow", "Vision"))
## [1] "Iron Man"        "Captain America" "Black Widow"     "Vision"

# DON'T
(avengers2 = c("Iron Man", "Captain America", "Black Widow", "Vision"))
## [1] "Iron Man"        "Captain America" "Black Widow"     "Vision"
```

On a final note, naming your objects is limited. You cannot chose any name. First, every name needs to start with a letter. Second, you can only use letters, numbers `_` and `.` as valid components of the names for your objects (see also Wickham and Grolemund, 2016, Chapter 4.2.). I recommend to establish a naming convention that you adhere to. Personally I prefer to only user lower letters and `_` to separate/connect words. You want to keep names informative, succinct and precise. Here are some examples of what some might consider good and bad choices for names.

```
# Good choices
income_per_annum
open_to_exp          # for 'openness to new experiences'
soc_int              # for 'social integration'

# Bad choices
IncomePerAnnum
measurement_of_boredom_of_watching_youtube
Sleep.per_monthsIn.hours
```

Ultimately, you need to be able to effectively work with your data and output. Ideally, this should be true for others as well who want or need to work with your R project as well, e.g. your co-investigator or supervisor. The same is true for your column names in datasets (see Chapter @ref()). Some more information about coding style (i.e. the style of writing coding) can be found in Chapter 5.5.

5.3 Functions

I used the term 'function' multiple times, but I never thoroughly explained what they are and why we need them. In simple terms, functions are objects. They contain lines of code that someone has written for us or we have written ourselves. One could say they are code snippets ready to use. Someone else might see them as shortcuts for our programming. Functions increase the speed with which we perform our analysis and write our computations and make our code more readable. Consider computing the `mean` of values stored in the object `pocket_money`.

```
# First we create an object that stores our desired values
pocket_money <- c(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89)

#1 Manually compute the mean
sum <- 0 + 1 + 1 + 2 + 3 + 5 + 8 + 13 + 21 + 34 + 55 + 89
sum / 12 # There are 12 items in the object
## [1] 19.33333

#2 Use a function to compute the mean
mean(pocket_money)
## [1] 19.33333

#3 Let's make sure #1 and #2 are actually the same
sum / 12 == mean(pocket_money)
## [1] TRUE
```

If we manually compute the mean, we first calculate the sum of all values in the object `pocket_money`[1]. Then we divide it by the number of values in the object, which is `12`. This is the traditional way of computing the mean as we know it from primary school. However, by simply using the function `mean()`, we not only write considerably less code, but it is also much easier to understand as well because the word `mean` does precisely what we would expect. Which one do you find easier?

To further illustrate how functions look like, let's create one ourselves and call it `my_mean`.

```
my_mean <- function(numbers){
  sum <- sum(numbers)               # Compute the sum of all values in 'numbers'
  result <- sum/length(numbers)     # Divide the sum by the number of items in 'numbers
  return(result)                    # Return the result in the console
}
```

[1] If you find the order of numbers suspicious, it is because it represents the famous Fibonacci sequence.

```
my_mean(pocket_money)
## [1] 19.33333
```

Do not worry if half of this code does not make sense to you. Writing functions is an advanced R skill. However, it is good to know how functions look on the 'inside'. You certainly can see the similarities between the code we have written before, but instead of using actual numbers, we work with placeholders like `numbers`. This way, we can use a function for different data and do not have to rewrite it every time.

All functions in R share the same structure. They have a `name` followed by `()`. Within these parentheses, we put `arguments`, which have specific `values`. For example, a function would look something like this:

```
name_of_function(argument_1 = value_1,
                 argument_2 = value_2,
                 argument_3 = value_3)
```

How many arguments there are and what kind of values you can provide is very much dependent on the function you use. Thus, not every function takes every value. In the case of `mean()`, the function takes an object which holds a sequence of `numeric` values. It would make very little sense to compute the mean of our `friends` object, because it only contains names. R would return an error message:

```
mean(friends)
## Warning in mean.default(friends): argument is not numeric or logical: returning
## NA
## [1] NA
```

`NA` refers to a value that is *'not available'*. In this case, R tries to compute the mean, but the result is not available, because the values are not `numeric` but a `character`. In your dataset, you might find cells that are `NA`, which means there is data missing. Remember: If a function attempts a computation that includes even just a single value that is `NA`, R will return `NA`. However, there is a way to fix this. You will learn more about how to deal with `NA` values in Chapter @ref().

Sometimes you will also get a message from R that states `NaN`. `NaN` stands for *'not a number'* and is returned when something is not possible to compute, for example:

```
# Example 1
0/0
## [1] NaN
```

```
# Example 2
sqrt(-9)
## Warning in sqrt(-9): NaNs produced
## [1] NaN
```

5.4 R packages

R has many built-in functions that we can use right away. However, some of the most interesting ones are developed by different programmers, data scientists and enthusiasts. To add more functions to your repertoire, you can install R packages. R packages are a collection of functions that you can download and use for your own analysis. Throughout this book, you will learn about and use many different R packages to accomplish various tasks.

To give you another analogy,

- R is like a global supermarket,
- RStudio is like my shopping cart,
- and R packages are the products I can pick from the shelves.

Luckily, R packages are free to use, so I do not have to bring my credit card. For me, these additional functions, developed by some of the most outstanding scientists, is what keeps me addicted to performing my research in R.

R packages do not only include functions but often include datasets and documentation of what each function does. This way, you can easily try every function right away, even without your own dataset and read through what each function in the package does. Figure 5.1

However, how do you find those R packages? They are right at your fingertips. You have two options:

1. Use the function `install.packages()`
2. Use the packages pane in RStudio (see Chapter 4.4)

5.4.1 Installing packages using `install.packages()`

The simplest and fastest way to install a package is calling the function `install.packages()`. You can either use it to install a single package or install a series of packages all at once using our trusty `c()` function. All you need to know is the name of the package. This approach works for all packages that are on CRAN (remember CRAN from Chapter 3.1?).

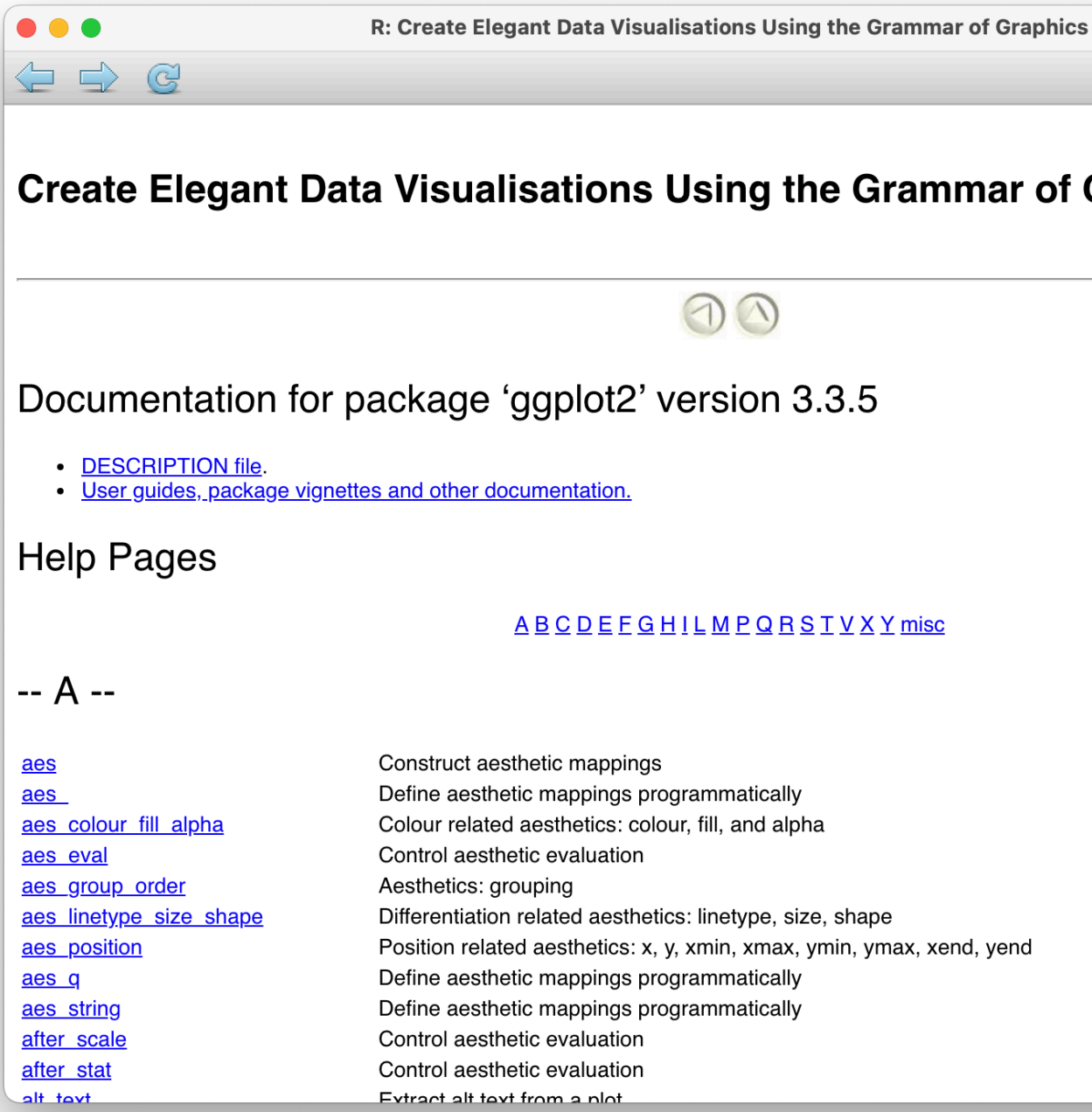


Figure 5.1: The R package documentation for 'ggplot2'

```
# Install a single package
install.packages("tidyverse")

# Install multiple packages at once
install.packages(c("tidyverse", "naniar", "psych"))
```

If a package is not available from CRAN, chances are you can find them on GitHub. GitHub is probably the world's largest global platform for programmers from all walks of life, and many of them develop fantastic R packages that make R programming not just easier but a lot more fun. As you continue to work in R, you should seriously consider creating your own account to keep backups of your R projects (see also Chapter 13.1).

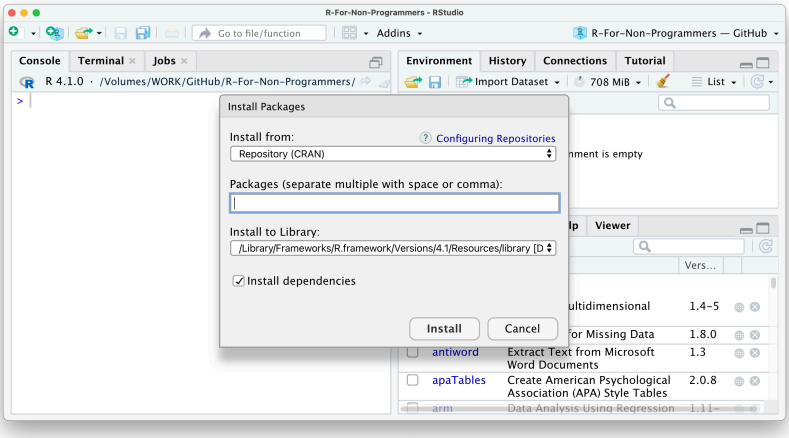
An essential companion for this book is `r4np`, which contains all datasets for this book and some useful functions to get you up and running in no time.

```
# Install the 'r4np' pacakge from GitHub
devtools::install_github("ddauber/r4np")
```

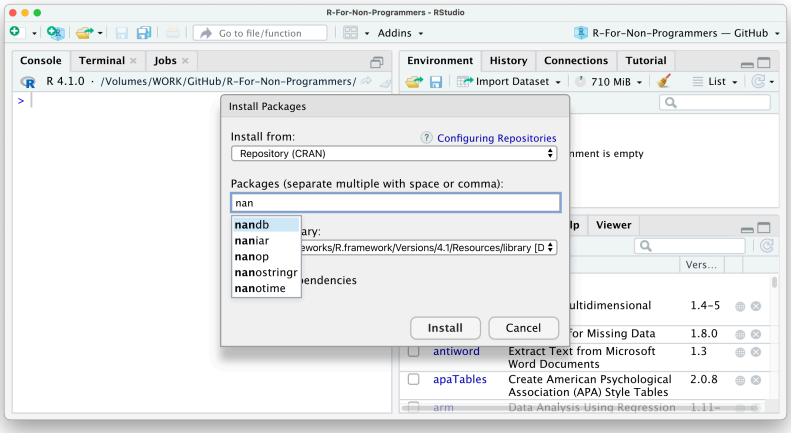
5.4.2 Installing packages via RStudio's package pane

RStudio offers a very convenient way of installing packages. In the packages pane, you cannot only see your installed packages, but you have two more buttons: `Install` and `Update`. The names are very self-explanatory. To install an R package you can follow the following steps:

1. Click on `Install`.
2. In most cases, you want to make sure you have `Repository (CRAN)` selected.

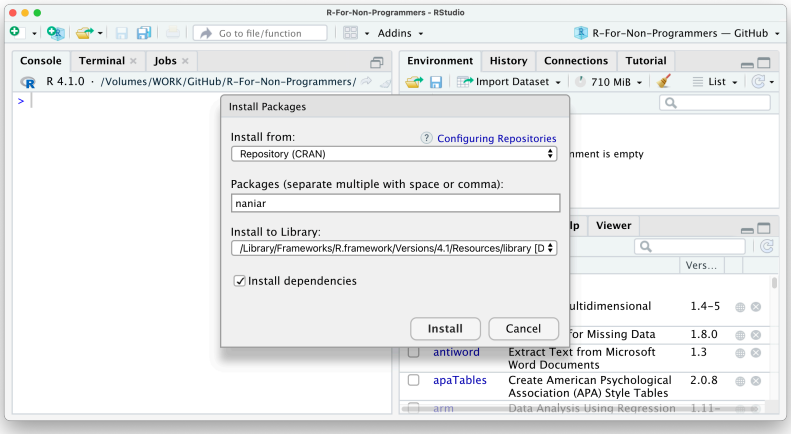


3. Type in the name of the package you wish to install. RStudio offers an auto-complete feature to make it even easier to find the package you want.



4. I recommend NOT to change the option which says `Install to library`. The default library settings will suffice.

5. Finally, I recommend to select `Install dependencies`, because some packages need other packages to function properly. This way, you do not have to do this manually.



The only real downside of using the packages pane is that you cannot install packages hosted on GitHub only. However, you can download them from there and install them directly from your computer using this option. This is particularly useful if you do not have an internet connection but you already downloaded the required packages onto a hard drive.

5.4.3 Using R Packages

Now that you have a nice collection of R packages, the next step would be to use them. While you only have to install R packages once, you have to 'activate' them every time you start an new session in RStudio. This process is also called 'loading an R package'. Once an R package is loaded, you can use all its functions. To load an R package, we have to use the function `library()`.

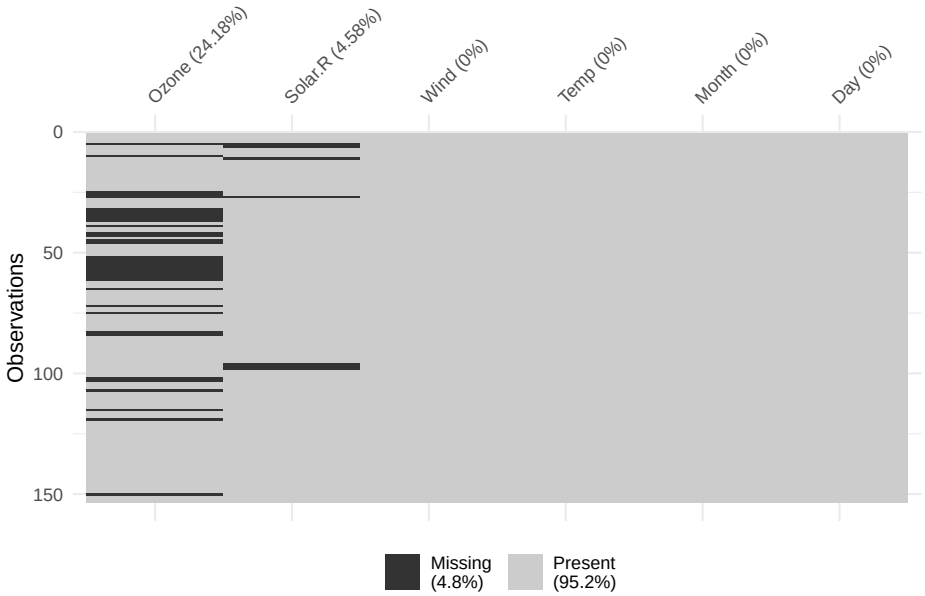
```
library(tidyverse)
```

The `tidyverse` package is a special kind of package. It contains multiple packages and loads them all at once. Almost all included packages (and more) you will use at some point when working through this book.

I know what you are thinking. Can you use `c()` to load all your packages at once? Unfortunately not. However, there is a way to do this, but it goes beyond the scope of this book to fully explain this (if you are curious, you can take a peek here).

Besides, it is not always advisable to load all functions of an entire package. One reason could be that two packages contain a function with the same name but with a different purpose. Two functions with the same name create a conflict between these two packages, and one of the functions would not be usable. Another reason could be that you only need to use the function once, and loading the whole package to use only one specific function seems excessive. Instead, you can explicitly call functions from packages without loading the package. For example, we might want to use the `vismis()` function from the `naniar` package to show where data is missing in our dataset `airquality`. Writing the code this way is also much quicker than loading the package and then calling the function if you don't use it repeatedly. Copy the code and try it yourself. Make sure you have `naniar` installed (see above). We will work with this package when we explore missing data in Chapter @ref().

```
# Here I use the dataset 'airquality', which comes with R
naniar::vis_miss(airquality)
```



5.5 Coding etiquette

Now you know everything to get started, but before we jump into our first project, I would like to briefly touch upon coding etiquette. This is not something that improves your analytical or coding skills directly, but is essential in building good habbits and making your life and those of others a little easier. Consider writing code like growing plants in your garden. You want to nurture the good plants, remove the weed and add labels that tell you which plant it is that you are growing. At the end of the day, you want your garden to be well-maintained. Treat you programming code the same way.

A script (see Chapter @ref()) with code should always have at least the following qualities:

- Only contains code that is necessary,
- Is easy to read and understand,
- Is self-contained.

With simple code this is easily achieved. However, what about more complex and longer code representing a whole set of analytical steps?

```
# Very messy code

library(tidyverse)
library(jtools)
model1 <- lm(covid_cases_per_1m ~ idv, data = df)
summ(model1, scale = TRUE, transform.response = TRUE, vifs = TRUE)
df %>% ggplot(aes(covid_cases_per_1m, idv, colour = europe, label = country))+
theme_minimal()+ geom_label(nudge_y = 2) + geom_point()
mod_model2 <- lm(cases_per_1m ~ idv + uai + idv*europe + uai*europe, data = df)
summ(mod_model2, scale = TRUE, transform.response = TRUE, vifs = TRUE)
anova(mod_model1, mod_model2)
```

How about the following in comparison?

```
# Nicely structured code

# Load required R packages
library(tidyverse)
library(jtools)

# ---- Modelling COVID-19 cases ----

## Specify and run a regression
model1 <- lm(covid_cases_per_1m ~ idv, data = df)

## Retrieve the summary statistics of model1
summ(model1, scale = TRUE, transform.response = TRUE, vifs = TRUE)

# Does is matter whether a country lies in Europe?

## Visualise rel. of covid cases, idv and being a European country
df %>%
  ggplot(aes(covid_cases_per_1m, idv, colour = europe, label = country))+
  theme_minimal()+
  geom_label(nudge_y = 2) +
  geom_point()

## Specify and run a revised regression
mod_model2 <- lm(cases_per_1m ~ idv + uai + idv*europe + uai*europe, data = df)

## Retrieve the summary statistics of model2
summ(mod_model2, scale = TRUE, transform.response = TRUE, vifs = TRUE)

## Test whether model2 is an improvement over model1
anova(mod_model1, mod_model2)
```

I hope we can agree that the second example is much easier to read and understand even though you probably do not understand most of it yet. For once, I separated the different analytical steps from each other like paragraphs in a report. Apart from that, I added comments with `#` to provide more context to my code for someone else who wants to understand my analysis. Admittedly, this example is a little excessive. Usually, you might have fewer comments. Commenting is an integral part of programming because it allows you to remember what you did. Ideally, you want to strike a good balance between commenting on and writing your code. How many comments you need will likely change throughout your R programming journey. Think of comments as headers for your programming script that give it structure..

We can use `#` not only to write comments but also to tell R not to run particular code. This is very helpful if you want to keep some code but do not want to use it yet. There is also a handy keyboard shortcut you can use to 'deactivate' multiple lines of code at once. Select whatever you want to 'comment out' in your script and press `Ctrl+Shift+C` (PC) or `Cmd+Shift+C` (Mac).

```
# mean(pocket_money) # R will NOT run this code
mean(pocket_money)   # R will run this code
```

RStudio helps a lot with keeping your coding tidy and properly formatted. However, there are some additional aspects worth considering. If you want to find out more about coding style, I highly recommend to read through the *'The tidyverse style guide'* (Wickham, 2021).

5.6 Exercises

1. What is the result of $\sqrt[2]{25 - 16} + 2 * 8 - 6$?
2. What does the console return if you execute the following code `"Five" == 5`?
3. Create a list called `books` and include the following book titles in it:
    - "Harry Potter and the Deathly Hallows",
    - "The Alchemist",
    - "The Davinci Code",
    - "R For Dummies"
4. Copy and paste the function below into your RStudio console and run it. What does the function do when you use it?

```r
x_x <- function(number1, number2){
  result1 <- number1*number2
  result2 <- sqrt(number1)
  result3 <- number1-number2
  return(c(result1, result2, result3))
}
```

5. What are the three steps to use a new R package that you found on CRAN?

Check you answers: Solutions 14.1

Chapter 6

Starting your R projects

Every project likely fills you with enthusiasm and excitement. And it should. You are about to find answers to your questions, and you hopefully come out more knowledgeable due to it. However, there are likely certain aspects of data analysis that you find less enjoyable. I can think of two:

- Keeping track of all the files my project generates
- Data wrangling

While we cover data wrangling in great detail later (Chapter 7), I would like to share some insights from my work that helped me stay organised and, consequently, less frustrated. The following applies to small and large research projects, which makes it very convenient no matter the situation. Of course, feel free to tweak my approach to whatever suits you. However, consistency is king.

6.1 Creating an R Project file

When working on a project, you likely create many different files for various purposes, especially R Scripts (see Chapter 6.3). If you are not careful, this file is stored in your system's default location, which might not be where you want them to be. RStudio allows you to manage your entire project intuitively and conveniently through R Project files. Using R Project files comes with a couple of perks, for example:

- All the files that you generate are in the same place. Your data, your coding, your exported plots, your reports, etc., all are in one place together without you having to manage the files manually.

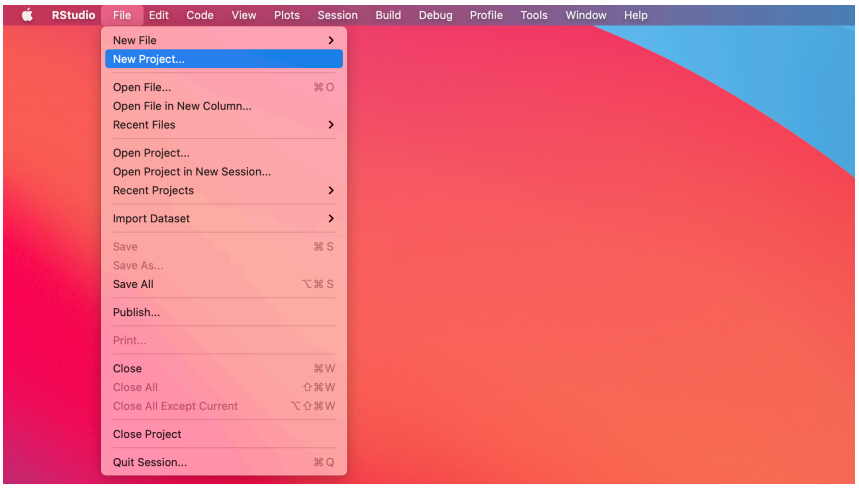
- If you want to share your project, you can share the entire folder, and others can quickly reproduce your research or help fix problems. This is because all file paths are relative and not absolute.

- You can, more easily, use GitHub for backups and so-called 'version control', which allows you to track changes you have made to your code over time (see also Chapter 13.1).

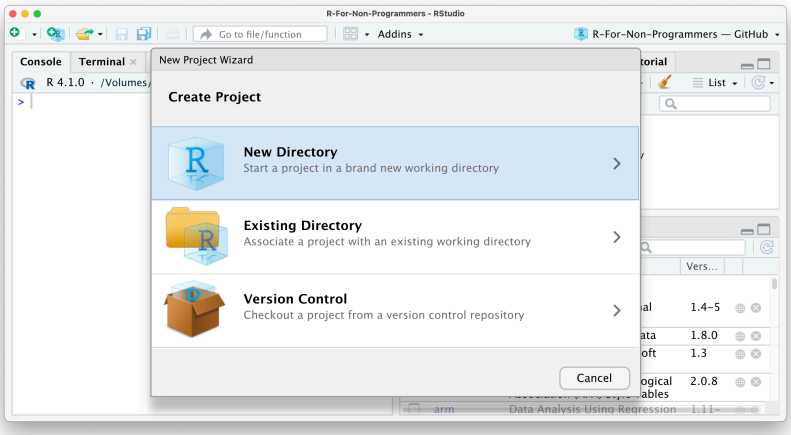
For now, the most important reason to use R Project files is the convenience of the organisation of files and the ability to share it easily with co-investigators, your supervisor, or your students.

To create an R Project, you need to perform the following steps:

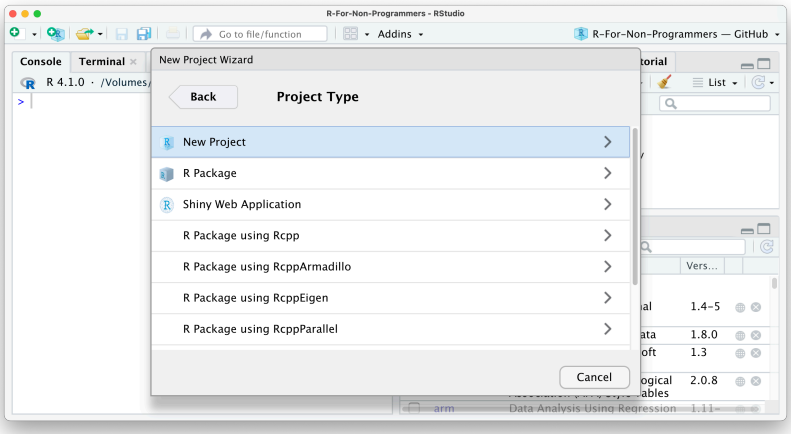
1. Select `File > New Project…` from the menu bar.



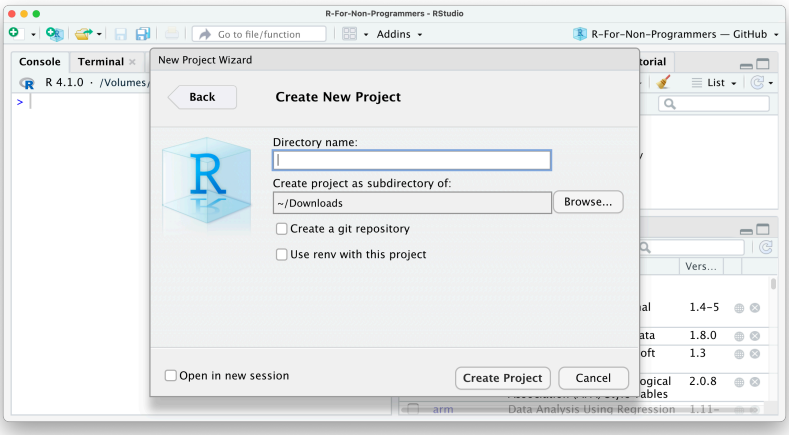
2. Select `New Directory` from the popup window.



3. Next, select `New Project`.

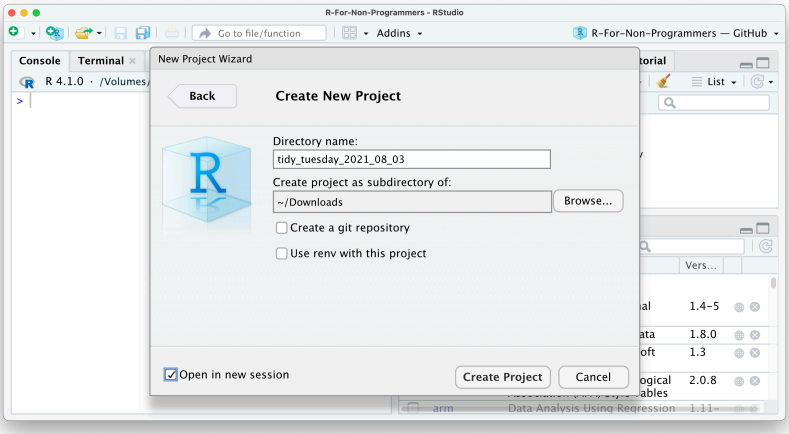


4. Pick a meaningful name for your project folder, i.e. the `Directory Name`. Ensure this project folder is created in the right place. You can change the `subdirectory` by clicking on `Browse...`. Ideally the subdirectory is a place where you usually store your research projects.

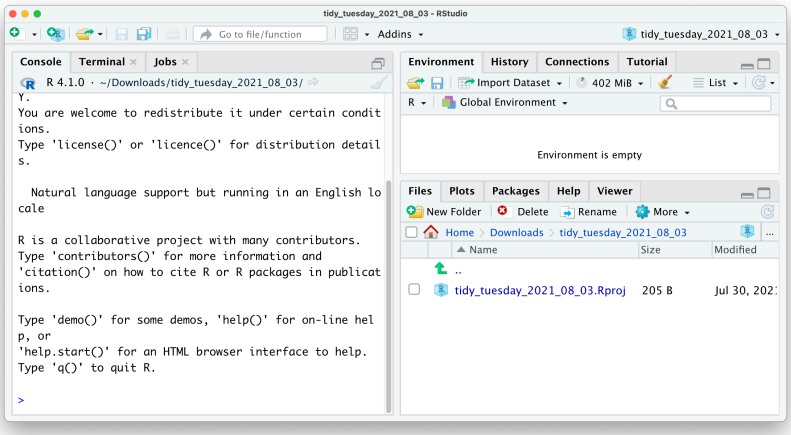


5. You have the option to `Create a git repository`. This is only relevant if you already have a GitHub account and wish to use version control. For now, you can happily ignore it.

6. Lastly, tick `Open in new session`. This will open your R Project in a new RStudio window.



7. Once you are happy with your choices, you can click `Create Project`. This will open a new R Session, and you can start working on your project.



If you look carefully, you can see that your RStudio is now 'branded' with your project name. At the top of the window, you see the project name, the files pane shows the root directory where all your files will be, and even the console shows on top the file path of your project. You could set all this up manually, but I would not recommend it, not the least because it is easy to work with R Projects.

6.2 Organising your projects

This section is not directly related to RStudio, R or data analysis in general. Instead, I want to convey to you that a good folder structure can go a long way. It is an excellent habit to start thinking about folder structures before you start working on your project. Placing your files into dedicated folders, rather than keeping them loosely in one container, will speed up your work and save you from the frustration of not finding the files you need. I have a template that I use regularly. You can either create it from scratch in RStudio or open your file browser and create the folders there. RStudio does not mind which way you do it. If you want to spend less time setting this up, you might want to use the function `create_dr()` from the `r4np` package. It creates all the folders as shown in Figure 6.1.

```
# Install 'r4np' from GitHub
devtools::install_github("ddauber/r4np")

# Create the template structure
r4np::create_dr()
```

To create a folder, click on `New Folder` in the Files pane. I usually have at least the following folders for every project I am involved in:

- A folder for my raw data. I store 'untouched' datasets in it. With 'untouched', I mean they have not been processed in any way and are usually files I downloaded from my data collection tool, e.g. online questionnaire platform.
- A folder with 'tidy' data. This is usually data I exported from R after cleaning it, i.e. after data wrangling (see Chapter 7).
- A folder for my R scripts
- A folder for my plots
- A folder for reports

Thus, in RStudio, it would look something like this:

You probably noticed that my folders have numbers in front of them. I do this to ensure that all folders are in the order I want them to be, usually not the alphabetical order my computer suggests. I use two digits because I may have more than nine folders for a project, and folder ten would otherwise be listed as the third folder in this list. With this filing strategy in place, it will be easy to find whatever I need. Even others can easily understand what I stored where. It is simply 'tidy', similar to how we want our data to be.

6.3 Creating an R Script

Code quickly becomes long and complex. Thus, it is not very convenient to write it in the console. So, instead, we can write code into an R Script. An R Script is a document that RStudio recognises as R programming code. Files that are not R Scripts, like `.txt`, `.rtf` or `.md`, can also be opened in RStudio, but any code written in it will not be automatically recognised.

When opening an R script or creating a new one, it will display in the source window (see Chapter 4.2). Some refer to this window as the 'script editor'. An R Script starts as an empty file. Good coding etiquette (see Chapter 5.5 demands that we use the first line to indicate what this file does by using a comment `#`. Here is an example for our 'TidyTuesday' R Project.

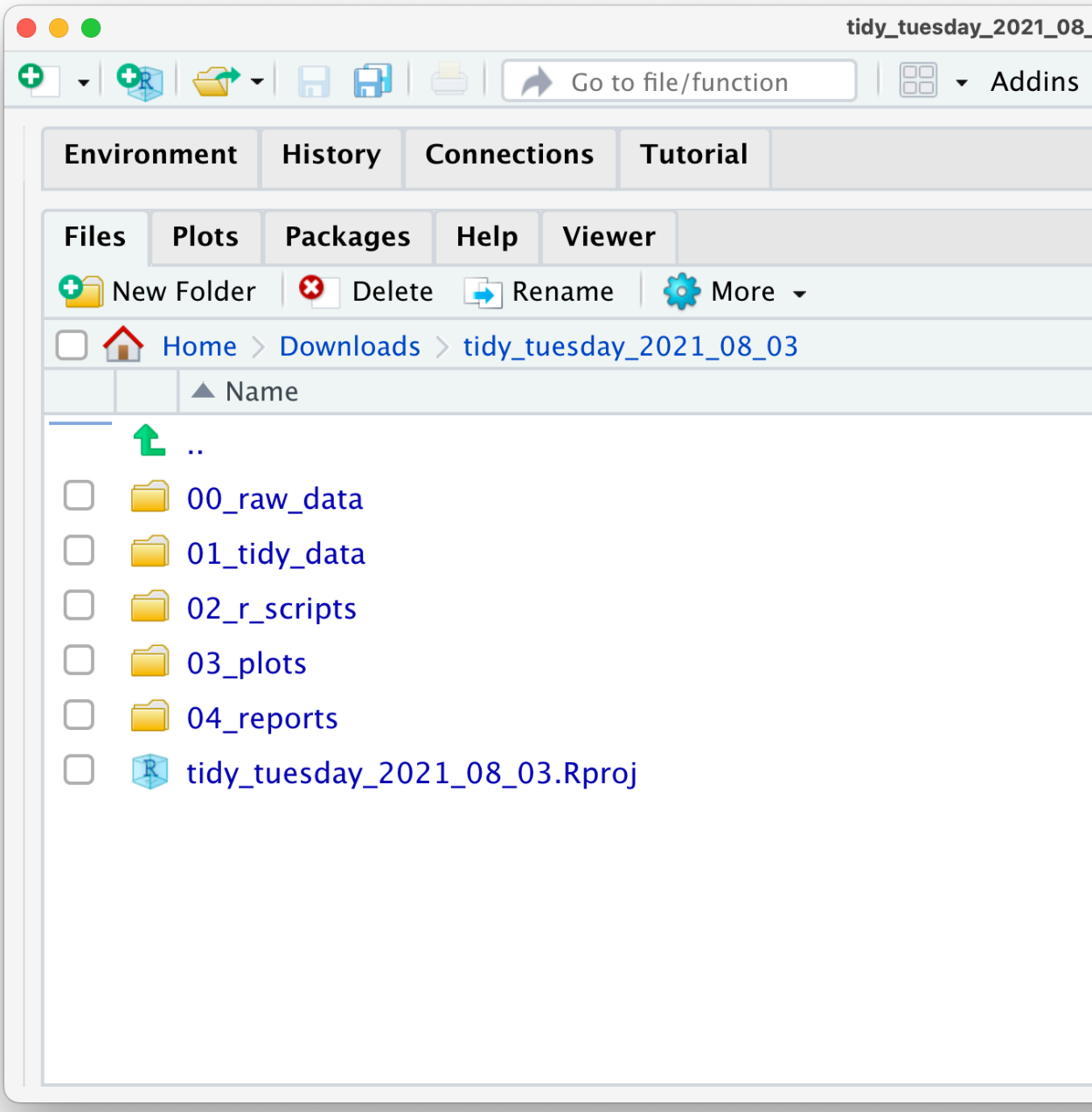
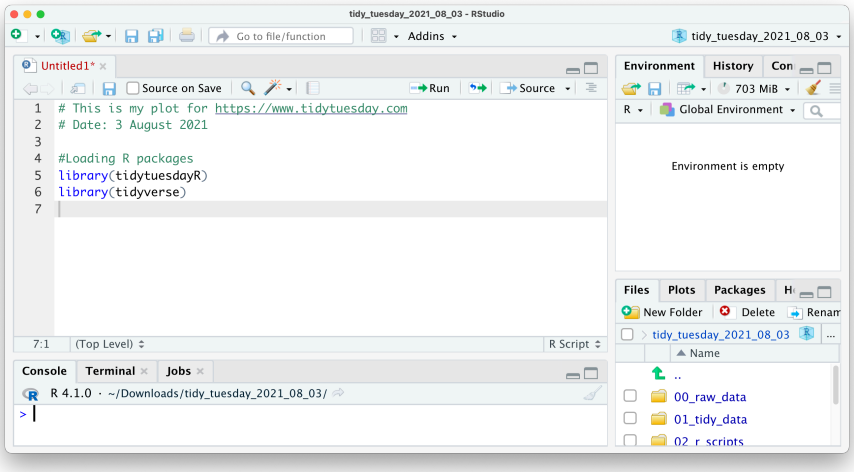
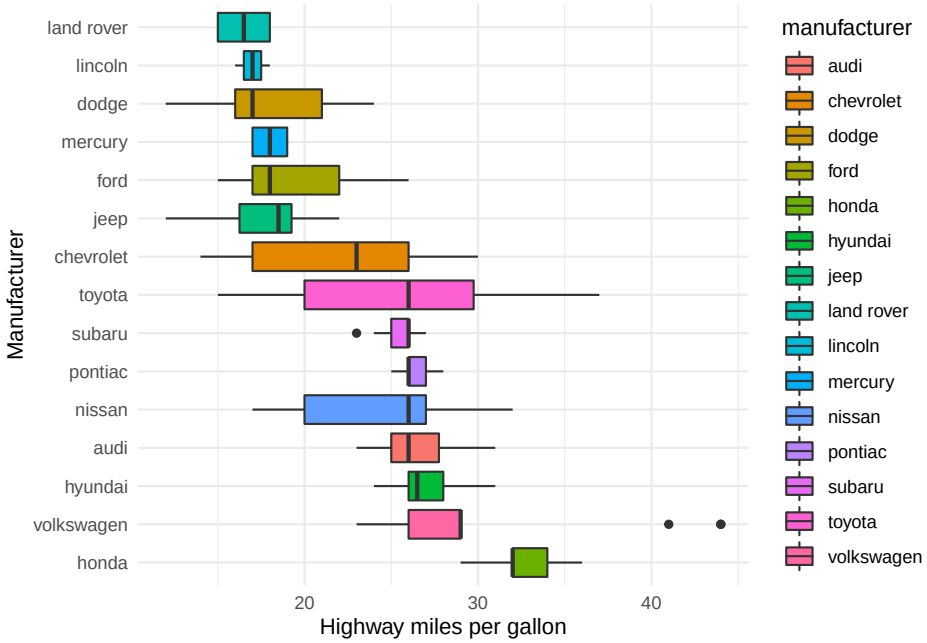


Figure 6.1: An example of a scalable folder structure for your project

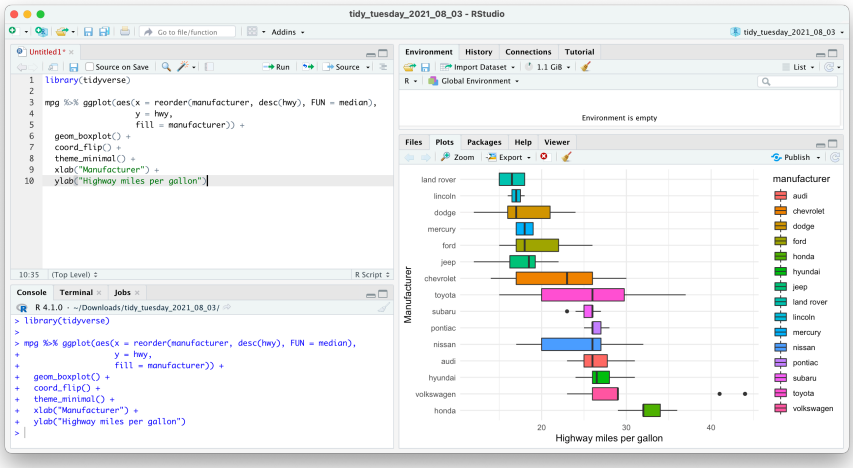


All examples in this book can easily be copied and pasted into your own R Script. However, for some code you will have to install the R package `r4np` (see above). Let's try it with the following code. The plot this code creates reveals which car manufacturer produces the most efficient cars.

```
library(tidyverse)

mpg %>% ggplot(aes(x = reorder(manufacturer, desc(hwy), FUN = median),
                   y = hwy,
                   fill = manufacturer)) +
  geom_boxplot() +
  coord_flip() +
  theme_minimal() +
  xlab("Manufacturer") +
  ylab("Highway miles per gallon")
```



You are probably wondering where your plot has gone. Copying the code will not automatically run it in your R Script. However, this is necessary to create the plot. If you tried pressing `Return` , you would only add a new line. Instead, you need to select the code you want to run and press `Ctrl+Return` (PC) or `Cmd+Return` (Mac). You can also use the `Run` command at the top of your source window, but it is much more efficient to press the keyboard shortcut. Besides, you will remember this shortcut quickly, because we need to use it very frequently. If all worked out, you should see the following:

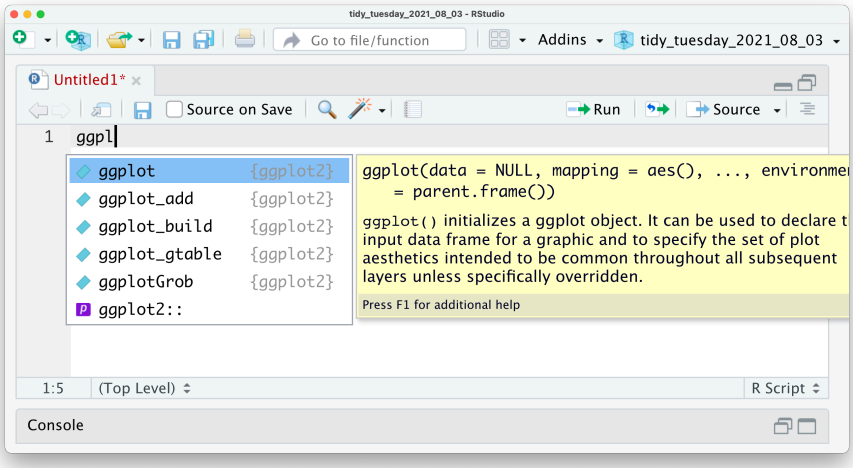


As you can see, cars from Honda appear to drive furthest with the same amount of fuel (a gallon) compared to other vehicles. Thus, if you are looking for a very economical car, you now know where to find them.

The R Script editor has some conveniences for writing your code that are worth pointing out. You probably noticed that some of the code we have pasted is blue, and some other code is in green. These colours help to make your code more readable because they carry a specific meaning. In the default settings, green stands for any values in `""`, which usually stands for `characters`. This is also called 'syntax highlighting'.

Moreover, code in R Scripts will be automatically indented to facilitate reading. If for whatever reason, the indentation does not happen, or you accidentally undo it, you can reindent a line with `Ctrl+I` (PC) or `Cmd+I` (Mac).

Lastly, the console and the R Script editor both feature code completion. This means that when you start typing a the name of function, R will provide suggestions. These are extremely helpful and make programming a lot faster. Once you found the function you were looking for, you press `Return` to insert it. Here is an example of what happens when you have the package `tidyverse` loaded and type `ggpl`. Only functions that are loaded via packages or any object in your environment pane benefit from code completion.



Not only does RStudio show you all the available options, but it also tells you which package this function is from. In this case, all listed functions are from the `ggplot2` package. Furthermore, when you select one of the options but have not pressed `Return` yet, you also get to see a yellow box, which provides you with a quick reference of all the arguments that this function accepts. So you do not have to memorise all the functions and their arguments.

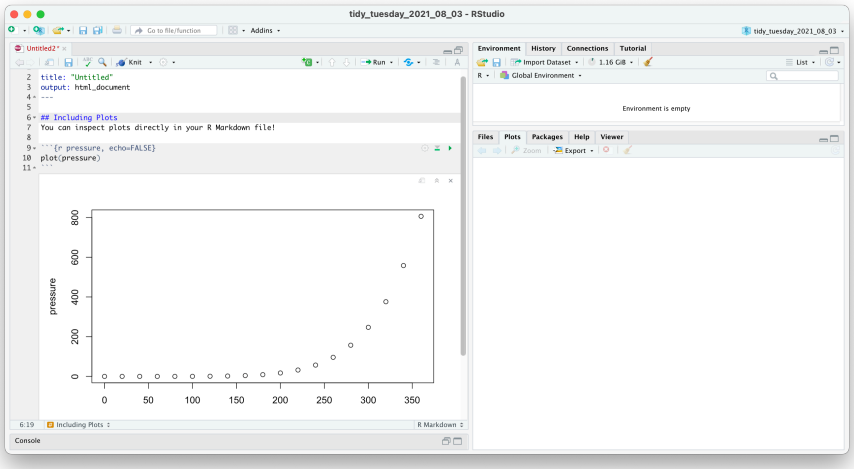
6.4 Using R Markdown

There is too much to say about R Markdown, which is why I only will highlight that they exist and point out the one feature that might convince you to choose these formats over plain R Scripts: They look like a Word document (almost).

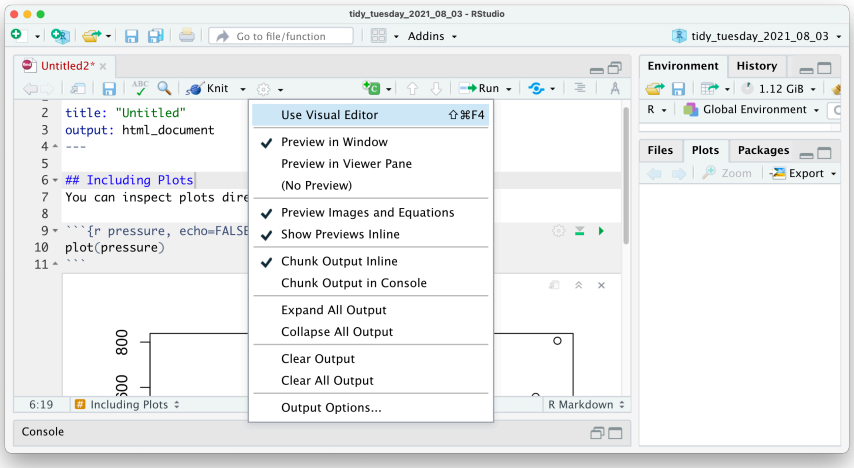
As the name indicates, R Markdown files are a combination of R Scripts and Markdown. Markdown is a way of writing and formatting text documents without needing software like MS Word. Instead, you write everything in plain text. Such plain text can be converted into many different document types such as HTML websites, PDF or Word documents. If you would like to see how it works, I recommend looking at the R Markdown Cheatsheet.

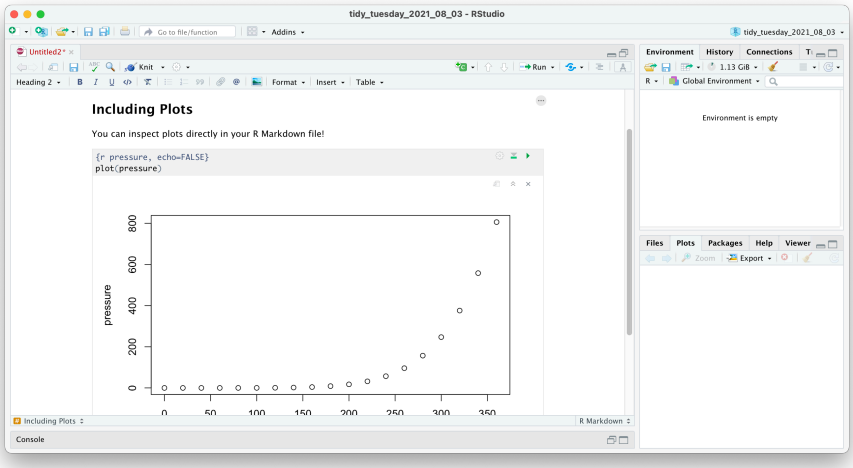
An R Markdown file works oppositely to an R Script. By default, an R Script considers everything as code and only through commenting `#` we can include text to describe what the code does. This is what you have seen in all the coding examples so far. On the other hand, an R Markdown file considers everything as text, and we have to specify what is code. We can do so by inserting 'code chunks'. Therefore, there is less of a need to use comments `#` in R Markdown files because you can write about it. Another convenience of R

Markdown files is that results from your analysis are immediately shown underneath the code chunk.



If you switch your view to the `Visual Editor`, it almost looks like you are writing a report in MS Word.





So, when should you use an R Script, and when should you use R Markdown. The rule-of-thumb is that if you intend to write a report, thesis or another form of publication, it might be better to work in an R Markdown file. If this does not apply, you might want to write an R Script. As mentioned above, R Markdown files emphasise text, while R Scripts primarily focus on code. In my projects, I often have a mixture of both. I use R Scripts to carry out data wrangling and my primary analysis and then use R Markdown files to present the findings, e.g. creating plots, tables, etc. By the way, this book is written in R Markdown using the `bookdown` package.

No matter your choice, it will neither benefit nor disadvantage you in your R journey or when working through this book. The choice is all yours. You likely will come to appreciate both formats for what they offer.

Chapter 7

Data Wrangling

You collected your data over months (and sometimes years), and all you want to know is whether your data makes sense and reveals something nobody would have ever expected. However, before we can truly go ahead with our analysis, it is essential to understand whether our data is 'tidy'. Very often, the data we receive is everything else but clean, and we need to check whether our data is fit for analysis and ensure it is in a format that is easy to handle. For small datasets, this is usually a brief exercise. However, I found myself cleaning data for a month because the dataset was spread out into multiple spreadsheets (no pun intended) with different numbers of columns and odd column names. Thus, data cleaning or data wrangling is an essential first step in any data analysis. It is a step that cannot be skipped and has to be performed on every new dataset.

Luckily, R provides many useful functions to make our lives easier. You will be in for a treat if you are like me and used to do this in Excel. It is a lot simpler using R to achieve a clean dataset.

Here is an overview of the different steps we usually work through before starting with our primary analysis. This list is certainly not exhaustive:

- Importing data
- Checking data types
- Recoding and arranging factors, i.e. categorical data.
- Running missing data diagnostics
- and other things

7.1 Import your data

The `r4np` package hosts several different datasets to work with, but at some point, you might want to apply your R knowledge to your own data. Therefore, an essential first step is to import your data into RStudio. There are three different methods, all of which are very handy:

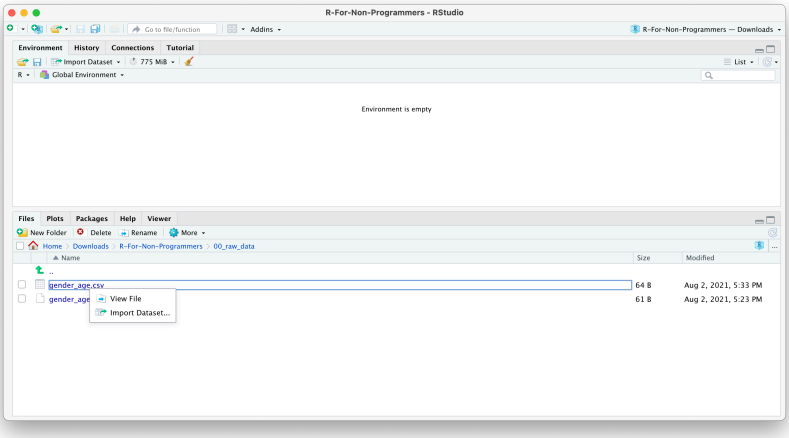
1. Click on your data file in the Files pane and choose `Import Dataset`.
2. Use the `Import Dataset` button in the Environment pane.
3. Import your data calling one of the `readr` functions in the console or RScript.

We will use the `readr` package to import our data. Using this package we can import a range of different file formats, including `.csv`, `.tsv`, `.txt`. If you want to import data from an `.xlsx` file, you need to use another package called `readxl`. The following sections will primarily focus on using `readr` via RStudio or directly in your Console or RScript.

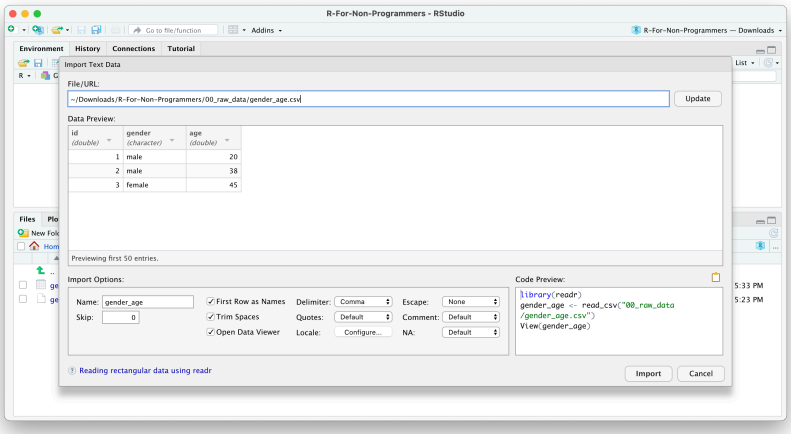
7.1.1 Import data from the Files pane

This approach is by far the easiest. Let's assume you have a dataset called `gender_age.csv` in your `00_raw_data folder`. If you wish to import it, you can do the following:

1. Click on the name of the file
2. Select `Import Dataset`.

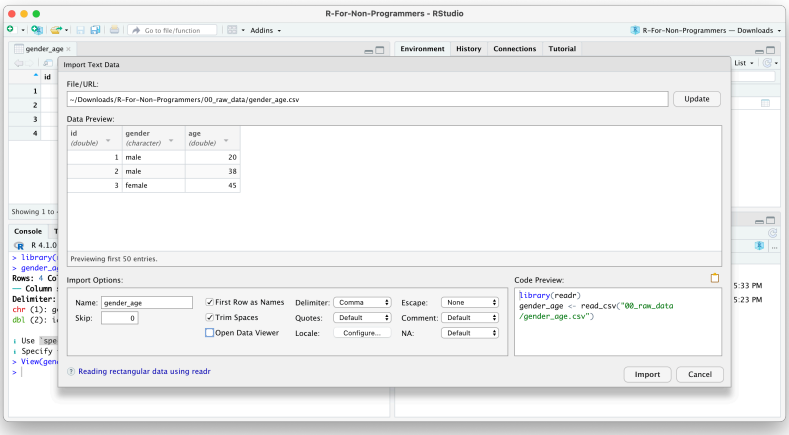


3. A new window will open, and you can choose different options. You also see a little preview of how the data looks like. This is great if you are not sure whether you did it correctly.

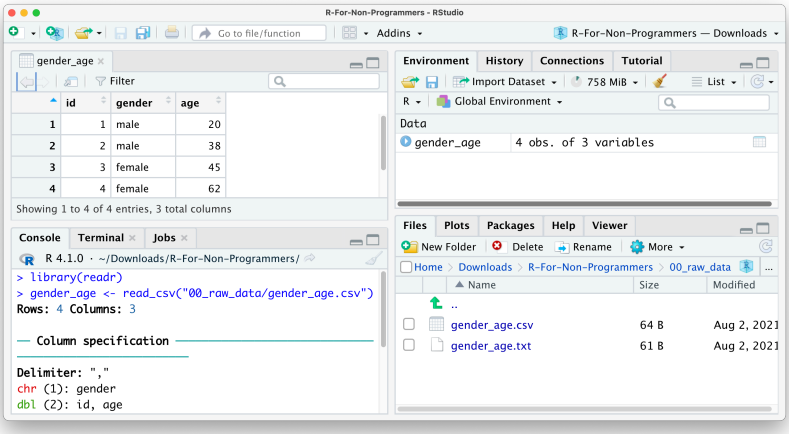


4. You can change how data should be imported, but the default should be fine in most cases. Here is a quick breakdown of the most important options:

   - `Name` allows you to change the object name, i.e. the name of the object this data will be assigned to. I often use `df_raw` (`df` stand for `data frame`, which is how R calls such rectangular datasets).
   - `Skip` is helpful if your data file starts with several empty rows at the top. You can remove them here.
   - `First Row as Names` is ticked by default. In most Social Science projects, we tend to have the name of the variables as the first row in your dataset.
   - `Trim Spaces` removes any unnecessary whitespace in your dataset. Leave it ticked.
   - `Open Data Viewer` allows you to look at your imported dataset. I use it rarely, but it can be helpful at times.
   - `Delimiter` defines how your columns are separate from each other in your file. If it is a `.csv` it would imply it is a 'comma-separated value', i.e. `,`. This setting can be changed for different files, depending on how your data is delimited. You can even use the option `Other…` to specify a custom separation option.
   - `NA` specifies how missing values in your data are acknowledged. By default, empty cells in your data will be recognised as missing data.



5. Once you are happy with your choices, you can click on `Import`.
6. You will find your dataset in the Environment pane.



In the Console, you can see that R also provides the `Column specification`, which we need later when inspecting 'data types'. `readr` automatically imports all text-based columns as `chr`, i.e. `character` values. However, this might not always be true. We will cover more of this aspect of data wrangling in Chapter 7.4.

7.1.2 Importing data from the Environment pane

The process of importing datasets from the Environment pane follows largely the one from the Files pane. Click on `Import Dataset > From Text`

(readr)…. The only main difference lies in having to find the file using the Browse… button. The rest of the steps are the same as above.

You will have to use the Environment pane for importing data from specific file types, e.g. .txt, because using the File pane would only open the file but not import the data for further processing.

7.1.3 Importing data using functions directly

If you organised your files well, it could be effortless and quick to use all the functions from readr directly. Here are two examples of how you can use readr to import your data. Make sure you have the tidyverse package loaded.

```
# Import data from '.csv'
read_csv("00_raw_data/gender_age.csv")

# Import data from any file text file by defining the separator yourself
read_delim("00_raw_data/gender_age.txt", delim = "|")
```

You might be wondering whether you can use read_delim() to import .csv files too. The answer is 'Yes, you can!'. In contrast to read_delim(), read_csv() sets the delimiter to , by default. This is mainly for convenience because .csv files are one of the most popular file formats used to store their data.

You might also be wondering what a 'delimiter' is. When you record data in a plain-text file, it is easy to see where a new observation starts and ends because it is defined by a row in your file. However, we also need to tell our software where a new column starts, i.e. where a cell begins and ends. Consider the following example. We have a file that holds our data which looks like this:

```
idagegender
124male
256female
333male
```

The first row we probably can still decipher as id, age, gender. However, the next row makes it difficult to understand which value represents the id of a participant and which value reflects the age of that participant. Like us, computer software would find it hard too to decide on this ambiguous content. Thus, we need to use delimiters to make it very clear which value belongs to which column. For example, In a .csv file, the data would be separated by a ,.

```
id,age,gender
1,24,male
2,56,female
3,33,male
```

Considering our example from above, we could also use `|` as a delimiter.

```
id|age|gender
1|24|male
2|56|female
3|33|male
```

There is a lot more to `readr` than could be covered in this book. If you want to know more about this R package, I highly recommend looking at the readr webpage.

7.2 Inspecting your data

For the rest of this chapter, we will use the `wvs` dataset from the `r4np` package. However, we do not know much about this dataset, and therefore we cannot ask any research questions worth investigating. Therefore we need to look at what it contains. The first method of inspecting a dataset is to type the name of the object, i.e. `wvs`.

```
# Ensure you loaded the 'r4np' package first
library(r4np)

# Show the data in the console
wvs
## # A tibble: 69,578 x 7
##    `Participant ID` `Country name` Gender   Age relationship_status
##               <dbl> <chr>           <dbl> <dbl> <chr>
##  1         20070001 Andorra             1    60 married
##  2         20070002 Andorra             0    47 living together as married
##  3         20070003 Andorra             0    48 separated
##  4         20070004 Andorra             1    62 living together as married
##  5         20070005 Andorra             0    49 living together as married
##  6         20070006 Andorra             1    51 married
##  7         20070007 Andorra             1    33 married
##  8         20070008 Andorra             0    55 widowed
##  9         20070009 Andorra             1    40 single
## 10         20070010 Andorra             1    38 living together as married
## # ... with 69,568 more rows, and 2 more variables: Freedom.of.Choice <dbl>,
## #   Satisfaction-with-life <dbl>
```

The result is a series of rows and columns. The first information we receive is: `A tibble: 69,578 x 9`. This indicates that our dataset has 69,578 observations (i.e. rows) and 9 columns (i.e. variables). This rectangular format is the one we encounter most frequently in Social Sciences (and probably beyond). If you ever worked in Microsoft Excel, this format will look familiar.

Even though it might be nice to look at a dataset in this way, it is not particularly useful. Depending on your monitor size, you might only see a small number of columns, and therefore we do not get to see a complete list of all variables. In short, we hardly ever will find much use in inspecting data this way. Luckily other functions can help us.

If you want to see each variable covered in the dataset and their data types, you can use the function `glimpse()` from the `dplyr` package (loaded as part of the `tidyverse` package).

```
glimpse(wvs)
## Rows: 69,578
## Columns: 7
## $ `Participant ID`         <dbl> 20070001, 20070002, 20070003, 20070004, 20070~
## $ `Country name`           <chr> "Andorra", "Andorra", "Andorra", "Andorra", "~
## $ Gender                   <dbl> 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, ~
## $ Age                      <dbl> 60, 47, 48, 62, 49, 51, 33, 55, 40, 38, 54, 3~
## $ relationship_status      <chr> "married", "living together as married", "sep~
## $ Freedom.of.Choice        <dbl> 10, 9, 9, 9, 8, 10, 10, 8, 8, 10, 9, 8, 10, 7~
## $ `Satisfaction-with-life` <dbl> 10, 9, 9, 8, 7, 10, 5, 8, 8, 10, 8, 8, 10, 7,~
```

The output of glimpse shows us the name of each column/variable after the `$`, for example, `` `Participant ID` ``. The `$` is used to lookup certain variables in our dataset. For example, if we want to inspect the column `relationship_status` only, we could write the following:

```
wvs$relationship_status
##     [1] "married"                   "living together as married"
##     [3] "separated"                 "living together as married"
##     [5] "living together as married" "married"
##     [7] "married"                   "widowed"
....
```

After the variable name, we find the recognised datatype for each column in `<...>`, for example `<chr>`. We will return to data types in Chapter 7.4. Lastly, we get samples of the data included. This output is much more helpful.

I use `glimpse()` very frequently for different purposes, for example:

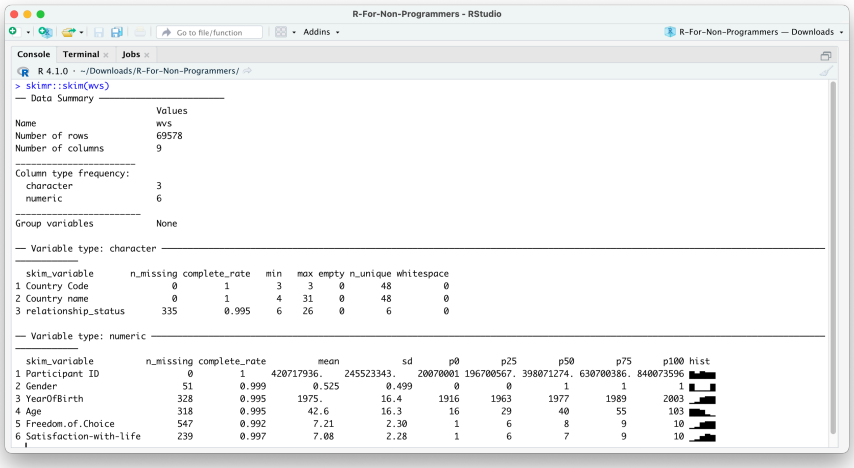
- to understand what variables are included in a dataset,

- to check the correctness of data types,
- to inspect variable names for typos or unconventional names,
- to look up variable names.

There is one more way to inspect your data and receive more information about it by using a specialised R package. The `skimr` package is excellent in 'skimming' your dataset. It provides not only information about variable names and data types but also provides some descriptive statistics. If you installed the `r4np` package and called the function `install_r4np()`, you will have `skimr` installed already.

```
skimr::skim(wvs)
```

The output in the Console should look like this:



As you can tell, there is a lot more information in this output. Many descriptive statistics that could be useful are already displayed. `skim()` provides a summary of the dataset and then automatically sorts the variables by data type. Depending on the data type, you also receive different descriptive statistics. As a bonus, the function also provides a histogram for numeric variables. However, there is one main problem: Some of the numeric variables are not numeric: `Participant ID` and `Gender`. Thus, we will have to correct the data types in a moment.

Inspecting your data in this way can be helpful to get a better understanding of what your data includes and spot problems with it. In addition, if you receive data from someone else, these methods are an excellent way to familiarise yourself with the dataset relatively quickly. Since I prepared this

particular dataset for this book, I also made sure to provide documentation for it. You can access it by using `?wvs` in the Console. This will open the documentation in the Help pane. Such documentation is available for every dataset we use in this book.

7.3 Cleaning your column names: Call the `janitor`

If you have an eagle eye, you might have noticed that most of the variable names in `wvs` are not consistent or easy to read/use.

```
# Whitespace and inconsistent capitalisation
Participant ID
Country name
Gender
Age

# Difficult to read
YearOfBirth
Freedom.of.Choice
Satisfaction-with-life
```

From Chapter 5.5, you will remember that being consistent in writing your code and naming your objects is essential. The same applies, of course, to variable names. R will not break using the existing names, but it will save you a lot of frustration if we take a minute to clean the names and make them more consistent.

You are probably thinking: "This is easy. I just open the dataset in Excel and change all the column names." Indeed, it would be a viable and easy option, but it is not very efficient, especially with larger datasets with many more variables. Instead, we can make use of the `janitor` package. By definition, `janitor` is a package that helps to clean up whatever needs cleaning. In our case, we want to tidy our column names. We can use the function `clean_names()` to achieve this. We store the result in a new object called `wvs` to keep those changes. The object will also show up in our Environment pane.

```
wvs <- janitor::clean_names(wvs)
glimpse(wvs)
## Rows: 69,578
## Columns: 7
## $ participant_id         <dbl> 20070001, 20070002, 20070003, 20070004, 2007000~
## $ country_name           <chr> "Andorra", "Andorra", "Andorra", "Andorra", "An~
## $ gender                 <dbl> 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0,~
```

```
## $ age                    <dbl> 60, 47, 48, 62, 49, 51, 33, 55, 40, 38, 54, 39,~
## $ relationship_status    <chr> "married", "living together as married", "separ~
## $ freedom_of_choice      <dbl> 10, 9, 9, 9, 8, 10, 10, 8, 8, 10, 9, 8, 10, 7, ~
## $ satisfaction_with_life <dbl> 10, 9, 9, 8, 7, 10, 5, 8, 8, 10, 8, 8, 10, 7, 1~
```

Now that `janitor` has done its magic, we suddenly have easy to read variable names that are consistent with the 'Tidyverse style guide' (Wickham, 2021).

If for whatever reason, the variable names are still not looking the way you want, you can use the function `rename()` from the `dplyr` package to manually assign new variable names.

```
wvs <- wvs %>% rename(satisfaction = satisfaction_with_life,
                      country = country_name)
glimpse(wvs)
## Rows: 69,578
## Columns: 7
## $ participant_id      <dbl> 20070001, 20070002, 20070003, 20070004, 20070005, ~
## $ country             <chr> "Andorra", "Andorra", "Andorra", "Andorra", "Andor~
## $ gender              <dbl> 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0,~
## $ age                 <dbl> 60, 47, 48, 62, 49, 51, 33, 55, 40, 38, 54, 39, 44~
## $ relationship_status <chr> "married", "living together as married", "separate~
## $ freedom_of_choice   <dbl> 10, 9, 9, 9, 8, 10, 10, 8, 8, 10, 9, 8, 10, 7, 10,~
## $ satisfaction        <dbl> 10, 9, 9, 8, 7, 10, 5, 8, 8, 10, 8, 8, 10, 7, 10, ~
```

You are probably wondering what `%>%` stands for. This symbol is called a *'piping operator'* , and it allows us to chain multiple functions together by considering the output of the previous function. So, do not confuse `<-` with `%>%`. Each operator serves a different purpose. The `%>%` has become synonymous with the `tidyverse` approach to R programming and is the chosen approach for this book. Many functions from the `tidyverse` are designed to be chained together.

If we wanted to spell out what we just did, we could say:

1. `wvs <-`: We assigned whatever happened to the right of the assignment operator to the object `wvs`.
2. `wvs %>%`: We defined the dataset we want to use with the functions defined after the `%>%`.
3. `rename(satisfaction = satisfcation_with_life)`: We define a new name `satisfaction` for the column `satisfaction_with_life`. Notice that the order is `new_name = old_name`. Here we also use `=`. A rare occasion where it makes sense to do so.

Just for clarification, the following two lines of code accomplish the same task. The only difference lies that with `%>%` we could chain another function right after it. So, you could say, it is a matter of taste which approach you prefer. However, in later chapters, it will become apparent why using `%>%` is very advantageous.

```
# Renaming a column using '%>%'
wvs %>% rename(satisfaction_new = satisfaction)

# Renaming a column without '%>%'
rename(wvs, satisfaction_new = satisfaction)
```

Since you will be using the pipe operator very frequently, it is a good idea to remember the keyboard shortcut for it: `Ctrl+Shift+M` for PC and `Cmd+Shift+M` for Mac.

7.4 Data types: What are they and how can you change them

When we inspected our data, I mentioned that some variables do not have the correct data type. You might be familiar with different data types by classifying them as:

- *Nominal data*, which is categorical data of no particular order,
- *Ordinal data*, which is categorical data with a defined order, and
- *Quantitative data*, which is data that usually is represented by numeric values.

In R we have a slightly different distinction:

- `character` / `<chr>`: Textual data, for example the text of a tweet.
- `factor` / `<fct>`: Categorical data with a finite number of categories with no particular order.
- `ordered` / `<ord>`: Categorical data with a finite number of categories with a particular order.
- `double` / `<dbl>`: Numerical data with decimal places.
- `integer` / `<int>`: Numerical data with whole numbers only (i.e. no decimals).

- `logical` / `<lgl>`: Logical data, which only consists of values 'TRUE' and 'FALSE'.
- `date` / `date`: Data which consists dates, e.g. '2021-08-05'.
- `date-time` / `dttm`: Data which consists dates and times, e.g. '2021-08-05 16:29:25 BST'.

For a complete list of data types, I recommend looking at 'Column Data Types' (Müller and Wickham, 2021).

R has a more fine-grained categorisation of data types. The most important distinction, though, lies between `<chr>`, `<fct>`/`<ord>` and `<dbl>` for most datasets in the Social Sciences. Still, it is good to know what the abbreviations in your `tibble` mean and how they might affect your analysis.

Now that we have a solid understanding of different data types, we can look at our dataset and see whether `readr` classified our variables correctly.

```
glimpse(wvs)
## Rows: 69,578
## Columns: 7
## $ participant_id      <dbl> 20070001, 20070002, 20070003, 20070004, 20070005, ~
## $ country             <chr> "Andorra", "Andorra", "Andorra", "Andorra", "Andor~
## $ gender              <dbl> 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0,~
## $ age                 <dbl> 60, 47, 48, 62, 49, 51, 33, 55, 40, 38, 54, 39, 44~
## $ relationship_status <chr> "married", "living together as married", "separate~
## $ freedom_of_choice   <dbl> 10, 9, 9, 9, 8, 10, 10, 8, 8, 10, 9, 8, 10, 7, 10,~
## $ satisfaction        <dbl> 10, 9, 9, 8, 7, 10, 5, 8, 8, 10, 8, 8, 10, 7, 10, ~
```

`readr` did a great job in identifying all the numeric variables. However, by default, `readr` imports all variables that include text as `<chr>`. It appears, in our dataset, this is not entirely correct. The variables `countr_name`, `gender` and `relationship_status` specify a finite number of categories. Therefore they should be classified as a `factor`. The variable `participant_id` is represented by numbers, but its meaning is also rather categorical. We would not use the ID numbers of participants to perform additions or multiplications. This would make no sense. Therefore, it might be wise to turn them into a `factor`, even though we likely will not use it in our analysis and would make no difference. However, I am a stickler for those kinds of things, and I would include in it.

To perform the conversion, we need to use two new functions from `dplyr`:

- `mutate()`: Changes, i.e. 'mutates', a variable.
- `as_factor()`: Converts data from one type into a `factor`.

If we want to convert all variables in one go, we can put them into the same function, separated by a `,`.

```
wvs <- wvs %>%
  mutate(country = as_factor(country),
         gender = as_factor(gender),
         relationship_status = as_factor(relationship_status),
         participant_id = as_factor(participant_id)
  )

glimpse(wvs)
## Rows: 69,578
## Columns: 7
## $ participant_id      <fct> 20070001, 20070002, 20070003, 20070004, 20070005, ~
## $ country             <fct> "Andorra", "Andorra", "Andorra", "Andorra", "Andor~
## $ gender              <fct> 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0,~
## $ age                 <dbl> 60, 47, 48, 62, 49, 51, 33, 55, 40, 38, 54, 39, 44~
## $ relationship_status <fct> married, living together as married, separated, li~
## $ freedom_of_choice   <dbl> 10, 9, 9, 9, 8, 10, 10, 8, 8, 10, 9, 8, 10, 7, 10,~
## $ satisfaction        <dbl> 10, 9, 9, 8, 7, 10, 5, 8, 8, 10, 8, 8, 10, 7, 10, ~
```

The output in the console shows that we successfully performed the transformation and our data types are as we intended them to be. Mission accomplished.

7.5 Handling factors

7.5.1 Recoding factors

Another common problem we have to tackle when working with data is their representation in the dataset. For example, `gender` could be measured as `male` and `female`[1] or as `0` and `1`. R does not mind which way you represent your data, but some other software does. Therefore, when we import data from somewhere else, the values of a variable might not look the way we want. The practicality of having your data represented accurately as what they are, becomes apparent when you intend to create tables and plots.

For example, we might be interested in knowing how many participants in the `wvs` were `male` and `female`. The function `count()` from `dplyr` does precisely that.

[1] A sophisticated research project would likely not rely on a dichotomous approach to gender, but appreciate the diversity in this regard.

```
wvs %>% count(gender)
## # A tibble: 3 x 2
##   gender     n
##   <fct>  <int>
## 1 0      33049
## 2 1      36478
## 3 <NA>      51
```

Now we know how many people were `male` and `female` and how many did not disclose their `gender`. Or do we? The issue here is that you would have to know what the `0` and `1` stand for. Surely you would have a coding manual that gives you the answer, but it seems a bit of a complication. For `gender`, this might still be easy to remember, but can you recall the ID numbers for 48 countries?

It certainly would be easier to replace the `0`s and `1`s with their corresponding labels. This can be achieved with a simple function called `fct_recode()` from `forcats`. However, since we 'mutate' a variable into something else, we also have to use the `mutate()` function.

```
wvs <- wvs %>% mutate(gender = fct_recode(gender, "male" = "0", "female" = "1"))
```

If you have been following along very carefully, you might spot one oddity in this code: `"0"` and `"1"`. You likely recall that in Chapter 5, I mentioned that we use `""` for `character` values but not for numbers. So what happens if we run the code and remove `""`.

```
wvs %>% mutate(gender = fct_recode(gender, "male" = 0, "female" = 1))
## Error: Problem with `mutate()` column `gender`.
## i `gender = fct_recode(gender, male = 0, female = 1)`.
## x Each input to fct_recode must be a single named string. Problems at positions: 1,
```

The error message is easy to understand: `fct_recode()` only expects `strings` as input and not numbers. R recognises `0` and `1` as numbers, but `fct_recode()` converts a `factor` value into another `factor` value. To refer to a factor level (i.e. one of the categories in our factor), we have to use `""`. In other words, data types matter and are often a source of problems with your code. Thus, always pay close attention to it.

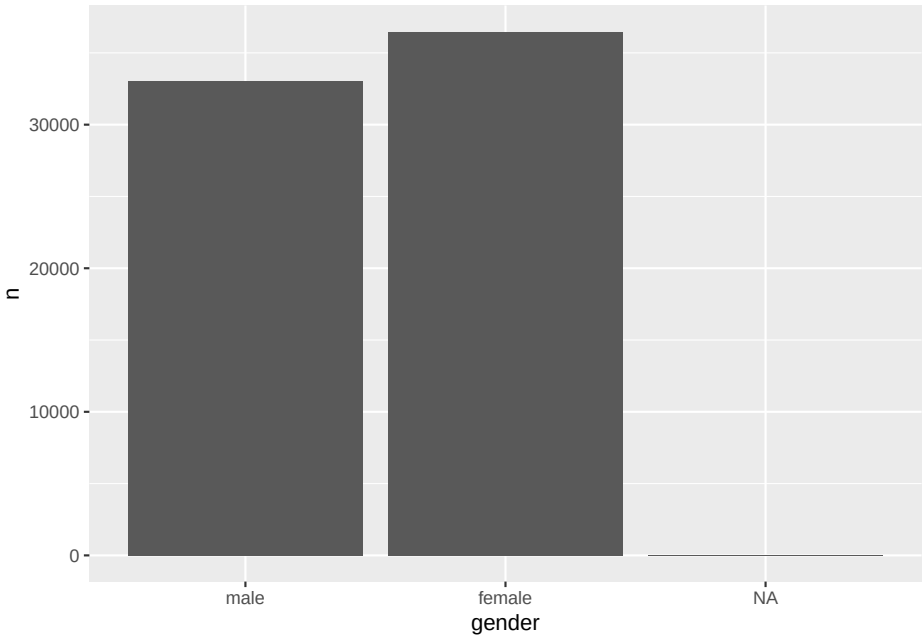
If we rerun our analysis and generate a frequency table for `gender`, we now get a much more readable output.

```
wvs %>% count(gender)
## # A tibble: 3 x 2
##   gender     n
```

```
##   <fct>  <int>
## 1 male   33049
## 2 female 36478
## 3 <NA>      51
```

Another benefit of going through the trouble of recoding your factors is the readability of your plots. For example, we could quickly generate a bar plot based on the above table and have appropriate labels instead of `0` and `1`.

```
wvs %>% count(gender) %>%
  ggplot(aes(gender, n)) +
  geom_col()
```



Plots are an excellent way to explore your data and understand relationships between variables. More about this when we start to perform analytical steps on our data (see Chapter 8 and beyond).

Another use case for recoding factors could be for purely cosmetic reasons. For example, when looking through our dataset, we might notice that some country names are very long and do not look great in data visualisations or tables. Thus, we could consider shortening them.

First, we need to find out which country names are particularly long. There are 48 countries in this dataset, so it could take some time to look through

them all. Instead, we could use the function `filter()` from `dplyr` to pick only countries with a long name. However, this poses another problem: How can we tell the filter function to pick only country names with a certain length? Ideally, we would want a function that does the counting for us. As you probably anticipated, there is a package called `stringr`, which also belongs to the `tidyverse`, and has a function that counts the number of characters that represent a value in our dataset: `str_length()`. This function takes any `character` variable and returns the length of it. This also works with `factors` because this function can 'coerce' it into a `character`, i.e. it just ignores that it is a `factor` and looks at it as if it was a regular `character` variable. Good news for us, because now we can put the puzzle pieces together.

```
wvs %>%
  filter(stringr::str_length(country) >= 15) %>%
  count(country)
## # A tibble: 3 x 2
##   country                              n
##   <fct>                            <int>
## 1 Bolivia, Plurinational State of   2067
## 2 Iran, Islamic Republic of         1499
## 3 Korea, Republic of                1245
```

I use the value `15` arbitrarily after some trial and error. You can change the value and see which other countries would show up with a lower threshold. However, this number seems to do the trick and returns three countries that seem to have longer labels. All we have to do is replace these categories with new ones the same way we recoded `gender`. You probably can guess already what we have to do to achieve this.

```
wvs <- wvs %>%
  mutate(country = fct_recode(country,
                              "Bolivia" = "Bolivia, Plurinational State of",
                              "Iran" = "Iran, Islamic Republic of",
                              "Korea" = "Korea, Republic of"))
```

7.5.2 Reordering factor levels

TODO: CONTINUE FROM HERE

- Consider whether this should happen here or later. Probably later, actually when we talk about descriptive statistics. This is not really data cleaning at this point. Too much stuff already. Move to descriptive statistics section.

7.6 Dealing with missing data

There is hardly any Social Sciences project where researchers do not have to deal with missing data. Participants are sometimes unwilling to complete a questionnaire or miss the second round of data collection entirely, e.g., longitudinal studies. It is not the purpose of this chapter to delve into all aspects of analysing missing data but provide a solid starting point. There are mainly three steps involved in dealing with missing data:

1. Mapping missing data
2. Identifying patterns of missing data
3. Replacing or removing missing data

7.6.1 Mapping missing data

Every study that intends to be rigorous will have to identify how much data is missing. In R, this can be achieved in multiple ways, but using a specialised package like `naniar` does help us to do this very quickly and systematically. First, we have to load the `naniar` package, and then we use the function `vis_miss()` to visualise how much and where exactly data is missing.

```
library(naniar)
vis_miss(wvs)
```

`naniar` plays along nicely with the `tidyverse` approach of programming. As such, it would also be possible to write `wvs %>% vis_miss()`.

As we can see, 99,7% of our dataset is complete, and we are only missing 0.3%. The dark lines (actually blocks) refer to missing data points. On the x-axis, we can see all our variables, and on the y-axis, we see our observations. This is the same layout as our rectangular dataset: Rows are observations, and columns are variables. Overall, this dataset appears relatively complete (luckily). In addition, we can see the percentage of missing data per variable. `freedom_of_choice` is the variable with the most missing data, i.e. 0.79%. Still, the amount of missing data is not very large.

When working with larger datasets, it might also be helpful to rank variables by their degree of missing data to see where the most significant problems lie.

```
gg_miss_var(wvs)
```

It is noticeable that `freedom_of_choice` has the most missing data points, while `participant_id`, and `country_name` have no missing values. If you

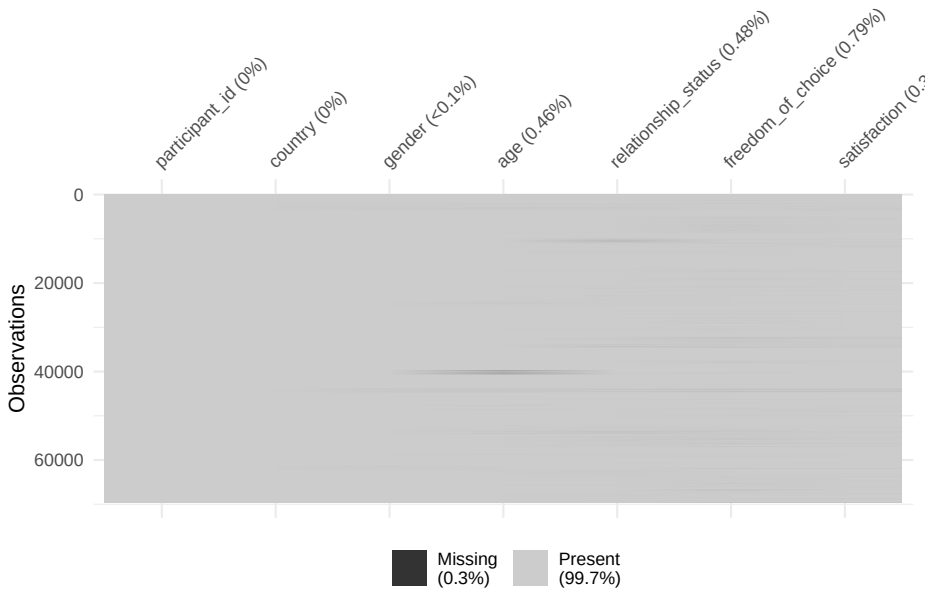


Figure 7.1: Mapping missing data with naniar

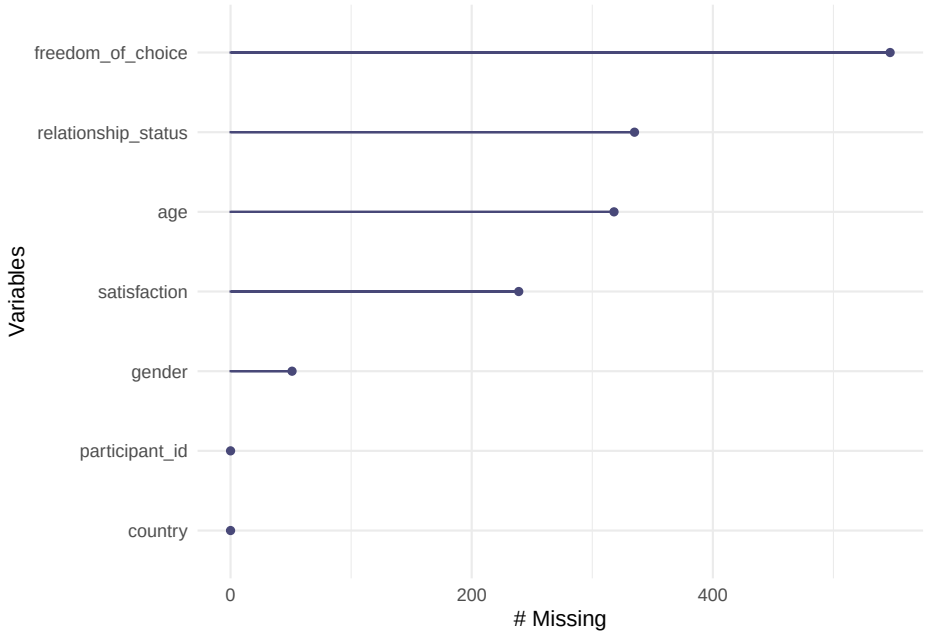


Figure 7.2: Missing data per variable

prefer to see the actual numbers instead, we can use a series of functions that start with `miss_` (for a complete list of all functions, see the reference page of naniar). For example, to retrieve the numeric values which are reflected in the plot above, we can write the following:

```
# Summarise the missingness in each variable
miss_var_summary(wvs)
## # A tibble: 7 x 3
##   variable            n_miss pct_miss
##   <chr>                <int>    <dbl>
## 1 freedom_of_choice      547   0.786
## 2 relationship_status    335   0.481
## 3 age                    318   0.457
## 4 satisfaction           239   0.343
## 5 gender                  51   0.0733
## 6 participant_id           0   0
## 7 country                  0   0
```

I tend to prefer data visualisations over numerical results for mapping missing data, especially in larger datasets with many variables. This also has the benefit that patterns of missing data can be more easily identified as well.

7.6.2 Identifying patterns of missing data

If you find that your data 'suffers' from missing data, it is essential to answer another question: Is data missing systematically? This is quite an important diagnostic step since systematically missing data would imply that if we remove these observations from our dataset, we likely produce the wrong results.

We can distinguish missing data based on how it is missing, i.e.

- missing completely at random (MCAR),
- missing at random (MAR), and
- missing not at random (MNAR). (Rubin, 1976)

7.6.2.1 Missing completely at random (MCAR)

Missing completely at random (MCAR) means that neither observed nor missing data can systematically explain why data is missing. It is a pure coincidence how data is missing, and there is no underlying pattern.

The `naniar` package comes with the very popular Little's MCAR test (Little, 1988), which provides insights into whether our data is missing completely at random. Thus, we can call the function `mcar_test()` and inspect the result.

```
wvs %>%
  select(-participant_id) %>%     # Remove variables which do not reflect a response
  mcar_test()
## # A tibble: 1 x 4
##   statistic    df p.value missing.patterns
##       <dbl> <dbl>   <dbl>            <int>
## 1      411.    67       0               19
```

When you run such a test, you have ensure that variables that are not part of the data collection are removed. In our case, the `participant_id` is generated by the researcher and does not represent an actual response by the participants. As such, we need to remove it using `select()` before we can run the test. A `-` inverts the meaning of `select()`. While `select(participant_id)` would do what it says, i.e. include it as the only variable in the test, `select(-participant_id)` results in selecting everything but this variable in our test. You will find it is sometimes easier to remove a variable with `select()` rather than listing all the variables you want to keep.

Since the `p.value` of the test is so small that it got rounded down to `0`, i.e. $p < 0.0001$, we have to assume that our data is not missing completely at random. If we found that $p > 0.05$, we would have confirmation that data are missing completely at random.

7.6.2.2 Missing at random (MAR)

Missing at random (MAR) refers to a situation where the observed data can explain missing data, but not the missing data. Dong and Peng (2013) (p. 2) provide a good example when this is the case:

> *Let's suppose that students who scored low on the pre-test are more likely to drop out of the course, hence, their scores on the post-test are missing. If we assume that the probability of missing the post-test depends only on scores on the pre-test, then the missing mechanism on the post-test is MAR. In other words, for students who have the same pre-test score, the probability of [them] missing the post-test is random.*

Thus,the main difference between MCAR and MAR data lies in the fact that we can observe some patterns of missing data if data is MAR. These patterns are only based on data we have, i.e. observed data. We also assume that no unobserved variables can explain these or other patterns.

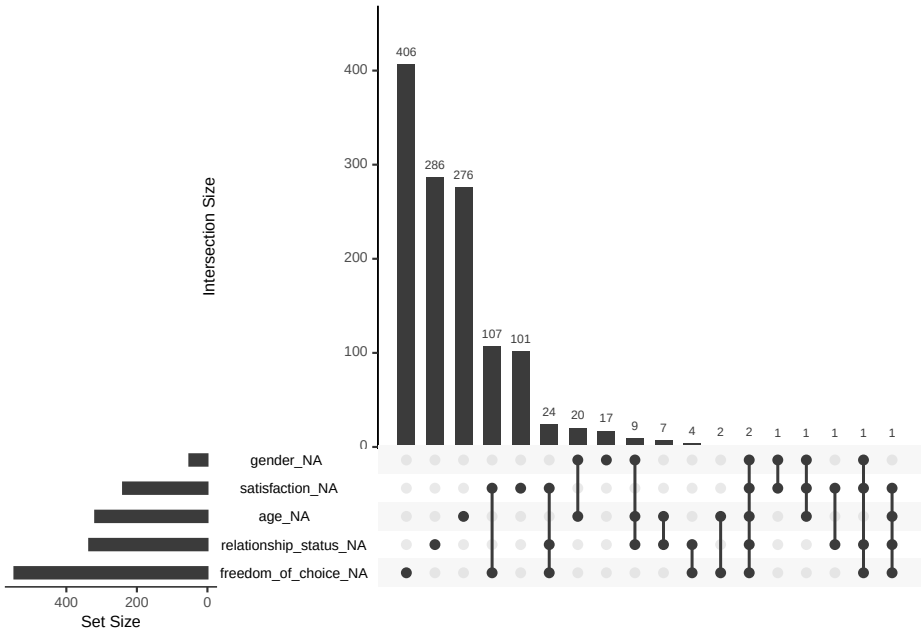
Accordingly, we first look into variables with no missing data and see whether they can explain our missing data in other variables. For example, we could investigate whether missing data in `freedom_of_choice` is attributed to specific countries.

```
wvs %>%
  group_by(country) %>%
  filter(is.na(freedom_of_choice)) %>%
  count() %>%
  arrange(desc(n))                        # Rearranging scores for easier reading
## # A tibble: 32 x 2
## # Groups:   country [32]
##    country         n
##    <fct>       <int>
##  1 Japan          47
##  2 Brazil         44
##  3 New Zealand    44
##  4 Russia         43
##  5 Bolivia        40
##  6 Romania        29
##  7 Kazakhstan     27
##  8 Turkey         24
##  9 Egypt          23
## 10 Serbia         20
## # ... with 22 more rows
```

It seems four countries have exceptionally high numbers of missing data for `freedom_of_choice`: Japan, Brazil, New Zealand, Russia and Bolivia. Why this is the case lies beyond this dataset and is something only the researchers themselves could explain. Collecting data in different countries is particularly challenging, and one is quickly faced with different unfavourable conditions. Furthermore, the missing data is not completely random because we have some first evidence that the location of data collection might have affected its completeness.

Another way of understanding patterns of missing data can be achieved by looking at relationships between missing values, for example, the co-occurrence of missing values across different variables. This can be achieved by using upset plots. An upset plot consists of three parts: Set size, intersection size and a Venn diagram which defines the intersections.

```
gg_miss_upset(wvs)
```



The most frequent combination of missing data in our dataset occurs when only `freedom_of_choice` is missing (the first column), but nothing else. Similar results can be found for `relationships_status` and `age`. The first combination of missing data is defined by two variables: `satisfaction` and `freedom_of_choice`. In total, 107 participants had `satisfaction` and `freedom_of_choice` missing but nothing else.

The 'set size' shown in the upset plot refers to the number of missing values for each variable in the diagram. This corresponds to what we have found when looking at Figure 7.2).

Our analysis also suggests that values are not completely randomly missing but that we have data to help explain why they are missing.

7.6.2.3 Missing not at random (MNAR)

Lastly, missing not at random (MNAR) implies that data is missing systematically and that other variables or reasons exist that explain why data is missing. Still, they are not fully known to us. In questionnaire-based research, an easily overlooked reason that can explain missing data is the 'page-drop-off' phenomenon. In such cases, participants stop completing a questionnaire once they advance to another page. Figure 7.3 shows this very clearly for a large scale project where an online questionnaire was used. After almost every page break in the questionnaire, some participants decided to discontinue. Finding these types of patterns is difficult when only working

with numeric values. Thus, it is always advisable to visualise your data as well. Such missing data is linked to the design of the data collection tool.

Defining whether a dataset is MNAR or not is mainly achieved by ruling out MCAR and MAR assumptions. It is not possible to test whether missing data is MNAR, unless we have more information about the underlying population available (van Ginkel et al., 2020).

We have sufficient evidence that our data is MAR as was shown above, because we managed to identify some relationships between unobserved and observed data. In practice, it is very rare to find datasets that are truly MCAR (van Buuren, 2018). Therefore we might consider 'imputation' as a possible strategy to solve our missing data problem. More about imputation in the next Chapter.

If you are looking for more inspiration of how you could visualise and identify patterns of missingness in your data, you might find the 'Gallery' of the `naniar` website particularly useful.

7.6.3 Replacing or removing missing data

Once you determined which pattern of missing data applies to your dataset, it is time to evaluate how we want to deal with those missing values. Generally, you can either keep the missing values as they are, replace them or remove them entirely.

Jakobsen et al. (2017) provide a rule of thumb of 5% where researchers can consider missing data as negligible, i.e. we can ignore the missing values because they won't affect our analysis in a significant way. They also argue that if the proportion of missing data exceeds 40%, we should also only work with our observed data. However, with such a large amount of missing data, it is questionable whether we can rely on our analysis as much as we want to. If the missing data lies somewhere in-between this range, we need to consider the missing data pattern at hand.

If data is MCAR, we could remove missing data. This process is called 'listwise deletion', i.e. you remove all data from a participant with missing data. As just mentioned, removing missing values is only suitable if you have relatively few missing values in your dataset (see also Schafer (1999)), as is the case with the `wvs` dataset. There are additional problems with deleting observations listwise, many of which are summarised by van Buuren (2018) in his introduction. Usually, we try to avoid removing data as much as possible.
If you wanted to perform listwise deletion, it can be done with a single function call: `na.omit()` from the built-in `stats` package. Here is an example of how we can apply this function.

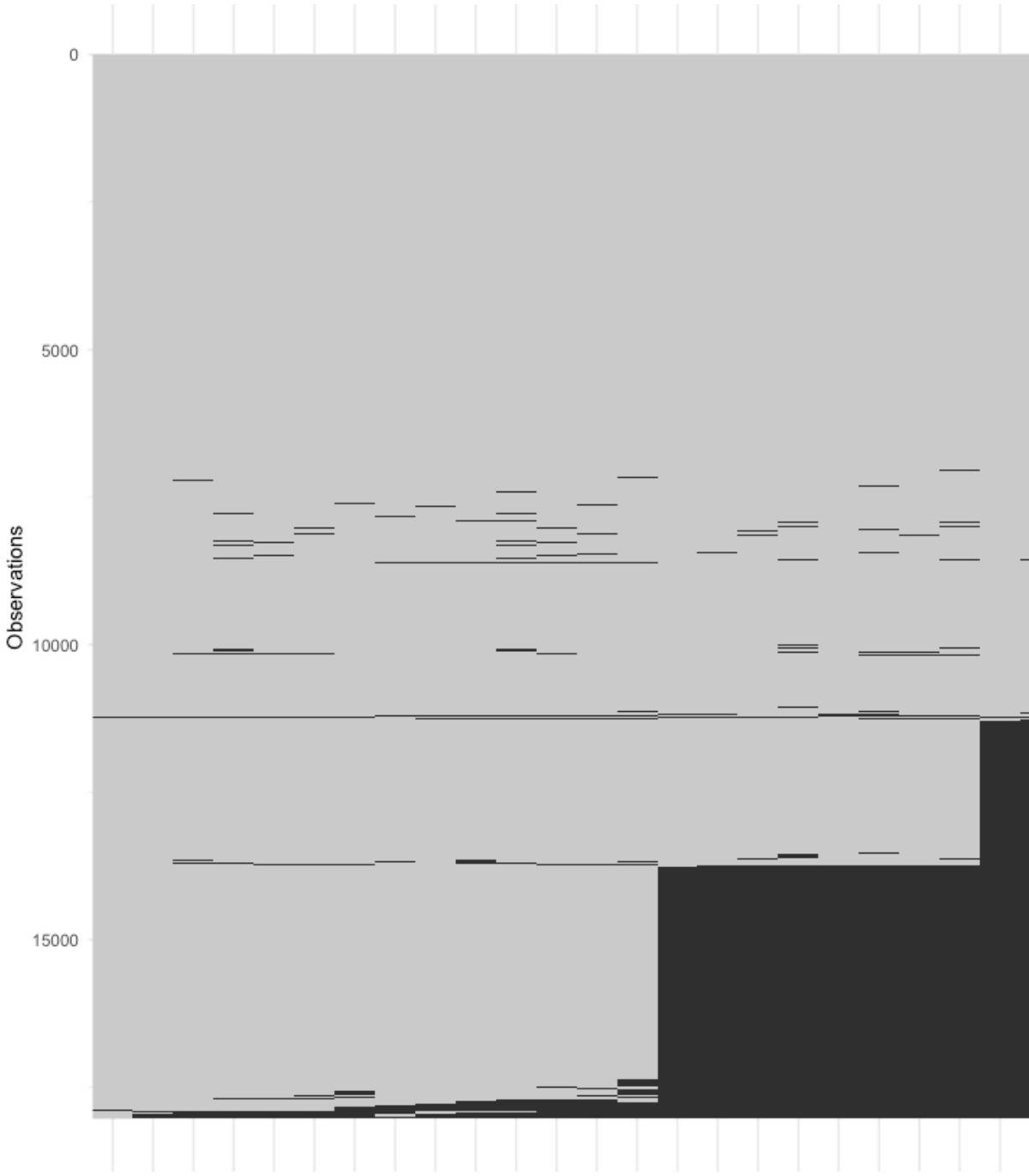
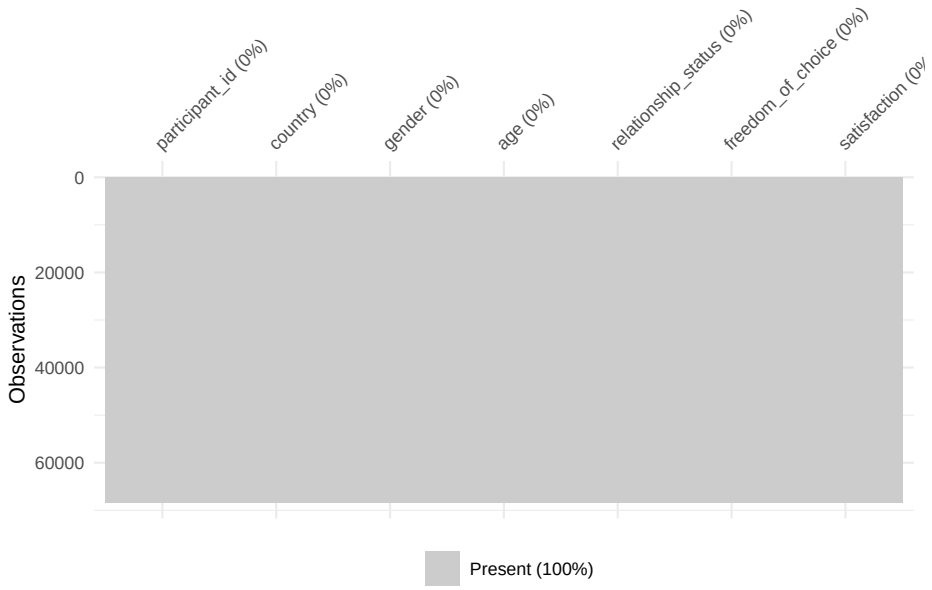


Figure 7.3: MNAR pattern in a dataset due to 'page-drop-offs'

```
# No more missing data in this plot
wvs %>% na.omit() %>% vis_miss()
```




```
# How much observations are left after we removed all missing data?
wvs %>% na.omit() %>% count()
## # A tibble: 1 x 1
##       n
##   <int>
## 1 68312
```

If you wanted to remove the missing data without the plot, you could use `wvs_no_na <- na.omit(wvs)`. However, I always recommend 'saving' the result in a new object to ensure I keep my original data available. This can be helpful when trying to compare how this decision affects my analysis, i.e. I can run the analysis with and without missing data removed. For the `wvs` dataset, this does not seem to be the best option.

Based on our analysis of the `wvs` dataset (by no means complete!), we could assume that data is MAR. In such cases, it is possible to 'impute' the missing values, i.e. replacing the missing data with computed scores. This is possible because we can model the missing data based on variables we have. We cannot model missing data based on variables we have not measured in our study for obvious reasons.

You might be thinking: "Why would it make data better if we 'make up' numbers? Is this not cheating?" Imputation of missing values is a science in

itself. There is plenty to read about the process of handling missing data, which would reach far beyond the scope of this book. However, the seminal work of van Buuren (2018), Dong and Peng (2013) and Jakobsen et al. (2017) are excellent starting points. In short: Simply removing missing data can lead to biases in your data, e.g. if we removed missing data in our dataset, we would mainly exclude participants from Japan, Brazil, New Zealand, Russia and Bolivia (as we found out earlier). While imputation is not perfect, scholars have shown that it can produce more reliable results than not using imputation at all (`add some references here`), assuming the data meets the requirements for such imputation.

Even though our dataset has only a minimal amount of missing data (relative to the entire size), we can still use imputation. There are many different ways to approach this task, one of which is 'multiple imputation'. As highlighted by van Ginkel et al. (2020), multiple imputation has not yet reached the same popularity as listwise deletion, despite its benefits. The main reason for this lies in the complexity of using this technique. Therefore I included an example of how to use multiple imputation separately in Chapter @ref(). There are many more approaches to imputation, and going through them all in detail would be impossible and distract from the book's main purpose. Still, I would like to share some interesting packages with you that use different imputation methods:

- mice (Multivariate Imputation via Chained Equations)
- Amelia (Uses a bootstrapped-based algorithm)
- missForest (Uses a random forest algorithm)
- Hmisc (Uses Additive Regression, Bootstrapping, and Predictive Mean Matching)
- mi (Uses posterior predictive distribution, predictive mean matching, mean-imputation, median-imputation, or conditional mean-imputation)

Besides multiple imputation, there is also the option for single imputation. The `simputation` package offers a great variety of imputation functions, one of which also fits our data quite well `impute_knn()`. This function makes use of a clustering technique called 'K-nearest neighbour'. In this case, the function will look for observations closest to the one with missing data and take the value of that observation. In other words, it looks for similar participants who answered the questionnaire in a very similar way. The great convenience of this approach is that it can handle all kinds of data types simultaneously, which is not true for all imputation techniques.

If we apply this function to our dataset, we have to write the following:

```
wvs_nona <- wvs %>%
  select(-participant_id) %>%
  as.data.frame() %>%                  # Transforms our tibble into a data frame
  simputation::impute_knn(. ~ .)

# Be aware that our new dataframe has different datatypes
glimpse(wvs_nona)
## Rows: 69,578
## Columns: 6
## $ country             <fct> Andorra, Andorra, Andorra, Andorra, Andorra, Andor~
## $ gender              <fct> female, male, male, female, male, female, female, ~
## $ age                 <chr> "60", "47", "48", "62", "49", "51", "33", "55", "4~
## $ relationship_status <fct> married, living together as married, separated, li~
## $ freedom_of_choice   <chr> "10", "9", "9", "9", "8", "10", "10", "8", "8", "1~
## $ satisfaction        <chr> "10", "9", "9", "8", "7", "10", "5", "8", "8", "10~

# Let's fix this
wvs_nona <- wvs_nona %>%
  mutate(age = as.numeric(age),
         freedom_of_choice = as.numeric(freedom_of_choice),
         satisfaction = as.numeric(satisfaction))
```

The function `impute_knn(. ~ .)` might look like a combination of text with an emoji `(. ~ .)`. This imputation function requires us to specify a model to impute the data. Since we want to impute all missing values in the dataset and use all variables available, we put `.` on both sides of the equation, separated by a `~`. The `.` reflects that we do not specify a specific variable but instead tell the function to use all variables that are not mentioned. In our case, we did not mention any variables at all, and therefore it chooses all of them. For example, if we wanted to impute only `freedom_of_choice`, we would have to put `impute_knn(freedom_of_choice ~ .)`. We will elaborate more on how to specify models when we cover regression models (see Chapter 11).

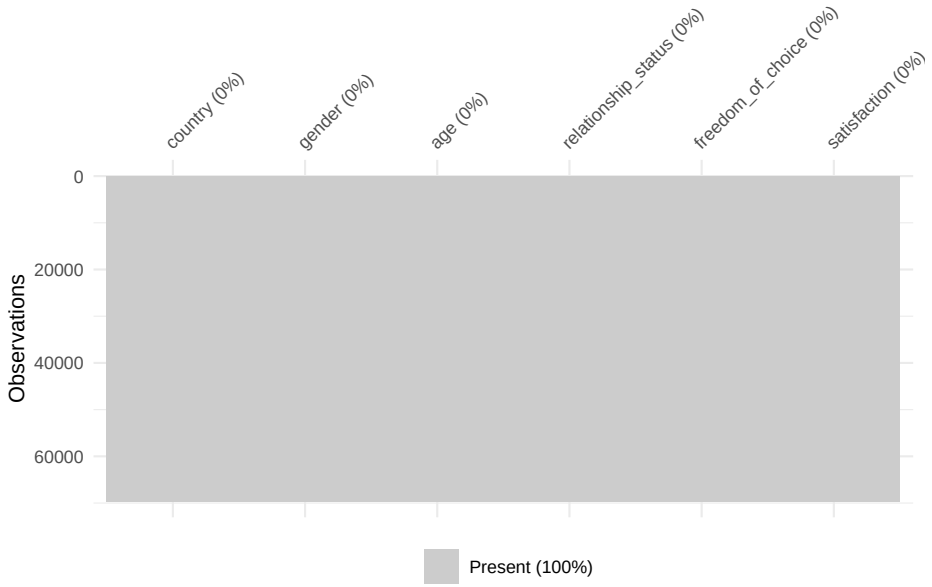
As you will have noticed, we also had to convert our `tibble` into a `data frame` using `as.data.frame()`. As mentioned in Chapter 5.3, some functions require a specific data type or format. The `simputation` package works with `dplyr`, but it prefers `data frame`s. Based on my experiences, the wrong data type or format is one of the most frequent causes of errors that novice R programmers report. So, keep an eye on it and read the documentation carefully. Also, once you inspect the new data frame, you will notice that some of our double variables have now turned into character values. Therefore, I strongly recommend checking your newly created data frames to avoid any surprises further down the line of your analysis.

Be aware that imputation of any kind can take a long time. For example, my MacBook Pro took about 4.22 seconds to complete `impute_knn()` with the

`wvs` dataset. If we used multiple imputation, this would have taken considerably longer, i.e. several minutes and more.

We now should have a dataset that is free of any missing values. To ensure this is the case we can create the missing data matrix that we made at the beginning of this chapter (see Figure 7.1).

```
vis_miss(wvs_nona)
```



7.6.4 Main takeaways regarding dealing with missing data

Handling missing data is hardly ever a simple process. Do not feel discouraged if you get lost in the myriad of options. While there is some guidance on how and when to use specific strategies to deal with missing values in your dataset, the most crucial point to remember is: Be transparent about what you did. As long as you can explain why you did something and how you did it, everyone can follow your thought process and help improve your analysis. However, ignoring the fact that data is missing and not acknowledging it is more than just unwise.

7.7 Latent constructs and their reliability

Social Scientists commonly face the challenge that we want to measure something that cannot be measured directly. For example, 'happiness' is a feeling that does not naturally occur as a number we can observe and measure. The opposite is true for 'temperature' which is naturally measured in numbers. At the same time, 'happiness' is much more complex of a variable than 'temperature', because 'happiness' can unfold in various ways and be caused by different triggers (e.g. a joke, an unexpected present, tasty food, etc.). To account for this, we often work with 'latent variables'. These are defined as variables that are not directly measured but are inferred from other variables. In practice, we often use multiple questions in a questionnaire to measure one latent variable, usually by computing the mean of those questions.

The `gep` dataset from the `r4np` package includes data about students' social integration experience (`si`) and communication skills development (`cs`). Data were obtained using the Global Education Profiler (GEP). I extracted 6 different questions (also called 'items') from the questionnaire, which measure `si`, and 6 items that measure `cs`. This dataset only consists of a randomly chosen set of responses, i.e. 300 out of over 12,000.

```
glimpse(gep)
## Rows: 300
## Columns: 12
## $ age                          <dbl> 22, 26, 21, 23, 25, 27, 24, 23, 21, 24, ~
## $ gender                       <chr> "Female", "Female", "Female", "Female", ~
## $ level_of_study               <chr> "UG", "PGT", "UG", "UG", "PGT", "PGT", "~
## $ si_socialise_with_people_exp <dbl> 6, 3, 4, 3, 4, 2, 5, 1, 6, 6, 3, 3, 4, 5~
## $ si_supportive_friends_exp    <dbl> 4, 4, 5, 4, 4, 2, 5, 1, 6, 6, 3, 3, 4, 6~
## $ si_time_socialising_exp      <dbl> 5, 2, 4, 4, 3, 2, 6, 3, 6, 6, 3, 2, 4, 6~
## $ cs_explain_ideas_imp         <dbl> 6, 5, 5, 5, 5, 5, 4, 5, 6, 6, 5, 5, 4, 6~
## $ cs_find_clarification_imp    <dbl> 4, 5, 5, 6, 6, 5, 4, 6, 6, 6, 5, 5, 4, 5~
## $ cs_learn_different_styles_imp <dbl> 6, 5, 6, 4, 4, 4, 4, 6, 6, 6, 4, 5, 5, 5~
## $ cs_explain_ideas_exp         <dbl> 6, 5, 2, 5, 6, 5, 6, 6, 6, 6, 4, 4, 3, 6~
## $ cs_find_clarification_exp    <dbl> 6, 5, 4, 6, 6, 5, 6, 6, 6, 6, 2, 5, 4, 5~
## $ cs_learn_different_styles_exp <dbl> 6, 4, 5, 4, 6, 3, 5, 6, 6, 6, 2, 4, 2, 5~
```

For example, if we wanted to know how each student scored with regards to social integration (`si`), we have to

- compute the mean (`mean()`) of all related items (i.e. all variables starting with `si_`),
- for each row (`rowwise()`) because each row presents one participant.

The same is true for communication skills (`cs_imp` and `cs_exp`). We can compute all three variables in one go. For each variable, we compute `mean()` and use `c()` to list all the variables that we want to include in the mean:

```
# Compute the scores for the latent variable 'si' and 'cs'
gep <- gep %>%
  rowwise() %>%
  mutate(si = mean(c(si_socialise_with_people_exp,
                     si_supportive_friends_exp,
                     si_time_socialising_exp
                     )
                   ),
         cs_imp = mean(c(cs_explain_ideas_imp,
                         cs_find_clarification_imp,
                         cs_learn_different_styles_imp
                         )
                       ),
         cs_exp = mean(c(cs_explain_ideas_exp,
                         cs_find_clarification_exp,
                         cs_learn_different_styles_exp
                         )
                       )
         )

glimpse(gep)
## Rows: 300
## Columns: 15
## Rowwise:
## $ age                           <dbl> 22, 26, 21, 23, 25, 27, 24, 23, 21, 24, ~
## $ gender                        <chr> "Female", "Female", "Female", "Female", ~
## $ level_of_study                <chr> "UG", "PGT", "UG", "UG", "PGT", "PGT", "~
## $ si_socialise_with_people_exp  <dbl> 6, 3, 4, 3, 4, 2, 5, 1, 6, 6, 3, 3, 4, 5~
## $ si_supportive_friends_exp     <dbl> 4, 4, 5, 4, 4, 2, 5, 1, 6, 6, 3, 3, 4, 6~
## $ si_time_socialising_exp       <dbl> 5, 2, 4, 4, 3, 2, 6, 3, 6, 6, 3, 2, 4, 6~
## $ cs_explain_ideas_imp          <dbl> 6, 5, 5, 5, 5, 5, 4, 5, 6, 6, 5, 5, 4, 6~
## $ cs_find_clarification_imp     <dbl> 4, 5, 5, 6, 6, 5, 4, 6, 6, 6, 5, 5, 4, 5~
## $ cs_learn_different_styles_imp <dbl> 6, 5, 6, 4, 4, 4, 4, 6, 6, 6, 4, 5, 5, 5~
## $ cs_explain_ideas_exp          <dbl> 6, 5, 2, 5, 6, 5, 6, 6, 6, 6, 4, 4, 3, 6~
## $ cs_find_clarification_exp     <dbl> 6, 5, 4, 6, 6, 5, 6, 6, 6, 6, 2, 5, 4, 5~
## $ cs_learn_different_styles_exp <dbl> 6, 4, 5, 4, 6, 3, 5, 6, 6, 6, 2, 4, 2, 5~
## $ si                            <dbl> 5.000000, 3.000000, 4.333333, 3.666667, ~
## $ cs_imp                        <dbl> 5.333333, 5.000000, 5.333333, 5.000000, ~
## $ cs_exp                        <dbl> 6.000000, 4.666667, 3.666667, 5.000000, ~
```

Compared to dealing with missing data, this is a fairly straightforward task. However, there is a caveat. Before we can compute the mean across all these

variables, we need to know and understand whether all these scores reliably contribute to one single latent variable. If not, we would be in trouble and make a significant mistake.

By far, the most common approach to assessing reliability (or more accurately 'internal consistency') of latent variables is Cronbach's α. This indicator looks at how strongly a set of items (i.e. questions in your questionnaire) are related to each other. For example, the stronger the relationship of all items starting with `si_` to each other, the more likely we achieve a higher Cronbach's α. The psych package has a suitable function to compute it for us.

Instead of listing all the items by hand, I use the function `starts_with()` to pick only the variables whose names start with `si_`. It certainly pays off to think about your variable names more thoroughly in advance to benefit from such shortcuts (see also Chapter 7.3.

```
gep %>%
  select(starts_with("si_")) %>%
  psych::alpha()
##
## Reliability analysis
## Call: psych::alpha(x = .)
##
##   raw_alpha std.alpha G6(smc) average_r S/N   ase mean  sd median_r
##       0.85      0.85     0.8      0.66 5.8 0.015  3.5 1.3     0.65
##
##  lower alpha upper     95% confidence boundaries
## 0.82 0.85 0.88
##
##  Reliability if an item is dropped:
##                              raw_alpha std.alpha G6(smc) average_r S/N alpha se
## si_socialise_with_people_exp      0.82      0.82    0.70      0.70 4.6    0.021
## si_supportive_friends_exp         0.77      0.77    0.63      0.63 3.4    0.027
## si_time_socialising_exp           0.79      0.79    0.65      0.65 3.7    0.025
##                              var.r med.r
## si_socialise_with_people_exp    NA  0.70
## si_supportive_friends_exp       NA  0.63
## si_time_socialising_exp         NA  0.65
##
##  Item statistics
##                                n raw.r std.r r.cor r.drop mean  sd
## si_socialise_with_people_exp 300  0.86  0.86  0.75   0.69  3.7 1.4
## si_supportive_friends_exp    300  0.90  0.89  0.81   0.75  3.5 1.6
## si_time_socialising_exp      300  0.88  0.88  0.79   0.73  3.2 1.5
##
## Non missing response frequency for each item
##                                1    2    3    4    5    6 miss
```

```
## si_socialise_with_people_exp 0.06 0.17 0.22 0.24 0.19 0.12    0
## si_supportive_friends_exp    0.13 0.18 0.20 0.18 0.18 0.13    0
## si_time_socialising_exp      0.15 0.21 0.22 0.20 0.12 0.10    0
```

The function `alpha()` returns a lot of information. The most important part, though, is shown at the very beginning:

```
##  raw_alpha std.alpha G6(smc) average_r  S/N  ase mean   sd median_r
##       0.85      0.85     0.8      0.66 5.76 0.01 3.46 1.33     0.65
```

In most publications, researchers would primarily report the `raw_alpha` value. This is fine, but it is not a bad idea to include at least `std.alpha` and `G6(smc)`.

In terms of interpretation, Cronbrach's α scores can range from 0 to 1. The closer the score to 1, the higher we would judge its reliability. Nunally (1967a) originally provided the following classification for Cronbach's α:

- between 0.6 and 0.5 can be sufficient during the early stages of development,
- 0.8 or higher is sufficient for most basic research,
- 0.9 or higher is suitable for applied research, where the questionnaires are used to make critical decisions, e.g. clinical studies, university admission tests, etc., with a 'desired standard' of 0.95.

However, a few years later, Nunally (1967b) revisited his original categorisation and considered 0.7 or higher as a suitable benchmark in more exploratory-type research. This gave grounds for researchers to pick and choose the 'right' publication for them (Henson, 2001). Consequently, depending on your research field, the expected reliability score might lean more towards 0.7 or 0.8. Still, the higher the score, the better it is.

Our dataset shows that `si` scores a solid $\alpha = 0.85$, which is excellent. We should repeat this step for `cs_imp` and `cs_exp` as well. However, we have to adjust `select()`, because we only want variables included that start with `cs_` and end with the facet of the respective variable, i.e. `_imp` or `_exp`. We also created two variables that start with `cs_`, which we also have to remove. Let me demonstrate the 'evolution' of how the `select()` function works to achieve what we want.

```
# Select only variables which start with 'cs_'
gep %>%
  select(starts_with("cs_")) %>%
```

```
  glimpse()
## Rows: 300
## Columns: 8
## Rowwise:
## $ cs_explain_ideas_imp          <dbl> 6, 5, 5, 5, 5, 5, 4, 5, 6, 6, 5, 5, 4, 6~
## $ cs_find_clarification_imp     <dbl> 4, 5, 5, 6, 6, 5, 4, 6, 6, 6, 5, 5, 4, 5~
## $ cs_learn_different_styles_imp <dbl> 6, 5, 6, 4, 4, 4, 4, 6, 6, 6, 4, 5, 5, 5~
## $ cs_explain_ideas_exp          <dbl> 6, 5, 2, 5, 6, 5, 6, 6, 6, 6, 4, 4, 3, 6~
## $ cs_find_clarification_exp     <dbl> 6, 5, 4, 6, 6, 5, 6, 6, 6, 6, 2, 5, 4, 5~
## $ cs_learn_different_styles_exp <dbl> 6, 4, 5, 4, 6, 3, 5, 6, 6, 6, 2, 4, 2, 5~
## $ cs_imp                        <dbl> 5.333333, 5.000000, 5.333333, 5.000000, ~
## $ cs_exp                        <dbl> 6.000000, 4.666667, 3.666667, 5.000000, ~

# Do the above but include only variables that also end with '_imp'
gep %>%
  select(starts_with("cs_") & ends_with("_imp")) %>%
  glimpse()
## Rows: 300
## Columns: 4
## Rowwise:
## $ cs_explain_ideas_imp          <dbl> 6, 5, 5, 5, 5, 5, 4, 5, 6, 6, 5, 5, 4, 6~
## $ cs_find_clarification_imp     <dbl> 4, 5, 5, 6, 6, 5, 4, 6, 6, 6, 5, 5, 4, 5~
## $ cs_learn_different_styles_imp <dbl> 6, 5, 6, 4, 4, 4, 4, 6, 6, 6, 4, 5, 5, 5~
## $ cs_imp                        <dbl> 5.333333, 5.000000, 5.333333, 5.000000, ~

# Remove 'cs_imp', because it is the computed latent variable
gep %>%
  select(starts_with("cs_") & ends_with("_imp"), -cs_imp) %>%
  glimpse()
## Rows: 300
## Columns: 3
## Rowwise:
## $ cs_explain_ideas_imp          <dbl> 6, 5, 5, 5, 5, 5, 4, 5, 6, 6, 5, 5, 4, 6~
## $ cs_find_clarification_imp     <dbl> 4, 5, 5, 6, 6, 5, 4, 6, 6, 6, 5, 5, 4, 5~
## $ cs_learn_different_styles_imp <dbl> 6, 5, 6, 4, 4, 4, 4, 6, 6, 6, 4, 5, 5, 5~
```

Since we want to compute the Cronbach's α for both variables, we write the following:

```
gep %>%
  select(starts_with("cs_") & ends_with("_imp"), -cs_imp) %>%
  psych::alpha()
##
## Reliability analysis
## Call: psych::alpha(x = .)
```

```
##
##   raw_alpha std.alpha G6(smc) average_r S/N   ase mean   sd median_r
##       0.86      0.87    0.82      0.68 6.5 0.014    5 0.98     0.65
##
##  lower alpha upper     95% confidence boundaries
## 0.84 0.86 0.89
##
##  Reliability if an item is dropped:
##                               raw_alpha std.alpha G6(smc) average_r S/N
## cs_explain_ideas_imp               0.78      0.78    0.65      0.65 3.6
## cs_find_clarification_imp          0.76      0.76    0.62      0.62 3.2
## cs_learn_different_styles_imp      0.88      0.88    0.79      0.79 7.4
##                               alpha se var.r med.r
## cs_explain_ideas_imp             0.025    NA  0.65
## cs_find_clarification_imp        0.028    NA  0.62
## cs_learn_different_styles_imp    0.014    NA  0.79
##
##  Item statistics
##                                 n raw.r std.r r.cor r.drop mean  sd
## cs_explain_ideas_imp          300  0.90  0.90  0.85   0.77  5.2 1.0
## cs_find_clarification_imp     300  0.91  0.91  0.87   0.79  5.1 1.1
## cs_learn_different_styles_imp 300  0.86  0.85  0.71   0.67  4.8 1.2
##
## Non missing response frequency for each item
##                                  1    2    3    4    5    6 miss
## cs_explain_ideas_imp          0.01 0.01 0.06 0.14 0.27 0.51    0
## cs_find_clarification_imp     0.01 0.02 0.05 0.14 0.30 0.47    0
## cs_learn_different_styles_imp 0.01 0.03 0.09 0.19 0.31 0.36    0
```

```
gep %>%
  select(starts_with("cs_") & ends_with("_exp"), -cs_exp) %>%
  psych::alpha()
```

```
##
## Reliability analysis
## Call: psych::alpha(x = .)
##
##   raw_alpha std.alpha G6(smc) average_r S/N   ase mean  sd median_r
##       0.81      0.82    0.76       0.6 4.4 0.019  4.2 1.1     0.58
##
##  lower alpha upper     95% confidence boundaries
## 0.78 0.81 0.85
##
##  Reliability if an item is dropped:
##                               raw_alpha std.alpha G6(smc) average_r S/N
## cs_explain_ideas_exp               0.69      0.69    0.53      0.53 2.3
```

```
## cs_find_clarification_exp          0.73      0.74    0.58      0.58 2.8
## cs_learn_different_styles_exp      0.81      0.81    0.68      0.68 4.2
##                                alpha se var.r med.r
## cs_explain_ideas_exp           0.035    NA  0.53
## cs_find_clarification_exp      0.031    NA  0.58
## cs_learn_different_styles_exp  0.022    NA  0.68
##
##  Item statistics
##                                  n raw.r std.r r.cor r.drop mean  sd
## cs_explain_ideas_exp           300  0.88  0.88  0.80   0.71  4.2 1.3
## cs_find_clarification_exp      300  0.85  0.86  0.76   0.68  4.5 1.2
## cs_learn_different_styles_exp 300  0.84  0.82  0.67   0.61  4.0 1.4
##
## Non missing response frequency for each item
##                                   1    2    3    4    5    6 miss
## cs_explain_ideas_exp           0.04 0.09 0.11 0.33 0.26 0.17    0
## cs_find_clarification_exp      0.02 0.05 0.13 0.27 0.32 0.21    0
## cs_learn_different_styles_exp 0.06 0.09 0.21 0.24 0.22 0.17    0
```

Similarly, to `si`, `cs_imp` and `cs_exp` show very good internal consistency scores too: $\alpha_{cs\_imp} = 0.86$ and $\alpha_{cs\_exp} = 0.81$. Based on these results we could be confident to use our latent variables for further analysis.

However, while Cronbach's α is very popular due to its simplicity, there is plenty of criticism (add references). Therefore, it is often not enough to report the Cronbach's α, but undertake additional steps. Depending on the stage of development of your measurement instrument (i.e. your questionnaire), you likely have to perform one of the following before computing the α scores:

- Exploratory factor analysis (EFA): Generally used to identify latent variables in a set of questionnaire items.
- Confirmatory factor analysis (CFA): To confirm whether a set of items truly reflect a latent variable.

An example of exploratory factor analysis is presented in Chapter @ref(). Since the `gep` data is based on an established measurement tool, we perform a CFA. To perform a CFA, we use the popular `lavaan` (Latent Variable Analysis) package. The steps of running a CFA in R include:

1. Define which variables are supposed to measure a specific latent variable (i.e. creating a model)
2. Run the CFA to see whether our model fits the data we collected
3. Interpret the results based on various indicators.

```
library(lavaan)
## This is lavaan 0.6-9
## lavaan is FREE software! Please report any bugs.

#1: Define the model which explains how items relate to latent variables
model <- '
social_integration =~
si_socialise_with_people_exp +
si_supportive_friends_exp +
si_time_socialising_exp

comm_skills_imp =~
cs_explain_ideas_imp +
cs_find_clarification_imp +
cs_learn_different_styles_imp

comm_skills_exp =~
cs_explain_ideas_exp +
cs_find_clarification_exp +
cs_learn_different_styles_exp
'

#2: Run the CFA to see how well this model fits our data
fit <- cfa(model, data = gep)

#3a: Extract the performance indicators
fit_indices <- fitmeasures(fit)

#3b: We tidy the results with the 'broom' package and pick only those indices we are m
broom::tidy(fit_indices) %>%
  filter(names == "cfi" |
         names == "srmr" |
         names == "rmsea") %>%
  mutate(x = round(x, 3))          # Round the results to 3 decimal places
## # A tibble: 3 x 2
##   names x
##   <chr> <lvn.vctr>
## 1 cfi   0.967
## 2 rmsea 0.081
## 3 srmr  0.037
```

The `broom` package is useful to clean output from all kinds of models, such as a CFA model. It allows us to convert the raw output into a `tibble`, which we can further manipulate using the functions we already know. If you want to know where `names` and `x` came from we have to inspect the output from

`broom::tidy(fit_indices)`.

```
broom::tidy(fit_indices)
## # A tibble: 42 x 2
##    names           x
##    <chr>           <lvn.vctr>
##  1 npar            2.100000e+01
##  2 fmin            1.177732e-01
##  3 chisq           7.066390e+01
##  4 df              2.400000e+01
##  5 pvalue          1.733810e-06
##  6 baseline.chisq  1.429729e+03
##  7 baseline.df     3.600000e+01
##  8 baseline.pvalue 0.000000e+00
##  9 cfi             9.665187e-01
## 10 tli             9.497780e-01
## # ... with 32 more rows
```

These column names were generated when we called the function `tidy()`. I often find myself working through chains of analytical steps iteratively to see what the intermediary steps produce. This also makes it easier to spot any mistake early on. Therefore, I recommend slowly building up your `dplyr` chains of function calls, especially when you just started learning R and the `tidyverse` approach of data analysis.

The results of our CFA appear fairly promising:

- The `cfi` (Comparative Fit Index) lies above 0.95 and
- the `srmr` (Standardised, Root Mean Square Residual) lies well below 0.08.
- The `rmsea` (Root Mean Square Error of Approximation) appears slightly higher than desirable, i.e. below 0.6. (Hu and Bentler, 1999)

Overall, however, the model seems to suggest a good fit with our data. Combined with the computed Cronbach's α, we can be reasonably confident in our latent variables and perform further analytical steps, for example, as shown in Chapter 11.

7.8 Once you finished with data wrangling

Once you finished your data cleaning, I recommend writing (i.e. exporting) your cleaned data out of R into your '01_tidy_data' folder. You can then use this dataset to continue with your analysis. This way, you do not have to run

all your data wrangling code every time you open your R project. To write your tidy dataset onto your hard drive of choice, we can use `readr` in the same as we did at the beginning. However, instead of `read_csv()` we have to use another function that writes the file into a folder. The function `write_csv()` first takes the object we want to save and then the folder and file name. We only made changes to the `wvs` dataset, so we should save it to our hard drive.

```
write_csv(wvs, "01_tidy_data/wvs_tidy.csv")
```

This chapter has been a reasonably long one. Nonetheless, it only covered the basics of what can and should be done when preparing data for analysis. These steps should not be rushed or even skipped. It is essential to have the data cleaned appropriately. This process helps you familiarise yourself with the dataset in great depth, and it makes you aware of limitations or even problems of your data. In the spirit of Open Science, using R also helps to document the steps you undertook to get from a raw dataset to a tidy one. This is also beneficial if you intend to publish your work later. Reproducibility has become an essential aspect of transparent and impactful research and should be a guiding principle of any empirical research.

Now that we have learned the basics of data wrangling (and there might be more waiting for you in the coming chapters), we can finally start our data analysis.

Chapter 8

Descriptive Statistics

The best way to understand how participants in your study have responded to various questions or an experimental treatment is to use descriptive statistics. As the name indicates, their main purpose is to 'describe'. Most of the time we want to describe the composition of our sample and likely how the majority (or minority) of participants performed.

In contrast, we use 'inferential' statistics to make predictions about the future. In Social Sciences, we are often interested in predicting how people will behave in certain situations and scenarios. We aim to develop models that help us navigate the complexity of social interactions that we all engage in, but might not fully understand. We cover 'inferential statistics' in later chapters of this book.

In short, descriptive statistics are an essential component to understand your data. To some extend, one could argue that we were already describing our data when we performed various data wrangling tasks (see Chapter 7.

The following chapters focus on the most essential descriptive statistics, i.e. the ones you likely want to investigate in 99 out of 100 research projects.

This book takes a 'visualise first' approach to data analysis. Therefore, each section will start with data visualisations, followed by computing statistical output, and end with how we can report our results in publications of all kind. A key learning outcome of this chapter is to plot your data using the package `ggplot2` and to present characteristics of your data in different ways. Each chapter will ask questions about our dataset that we aim to answer. However, first we need to understand how to create plots in R.

8.1 Plotting in R with `ggplot2`

Plotting can appear intimidating at first, but is very easy and quick, once you understand the basics. The `ggplot2` package is a very popular package to generate plots in R and many other packages are built upon it. This makes it a very flexible tool to create almost any data visualisation you could think of. If you want to see what is possible with `ggplot2` you might want to consider looking at #tidytuesday on Twitter where novices and veterans share their data visualisations on a weekly basis.

To generate any plot we need to at least define three components:

- a dataset,
- variables we want to plot, and
- how we want to plot them, e.g. as lines, bars, points, etc.

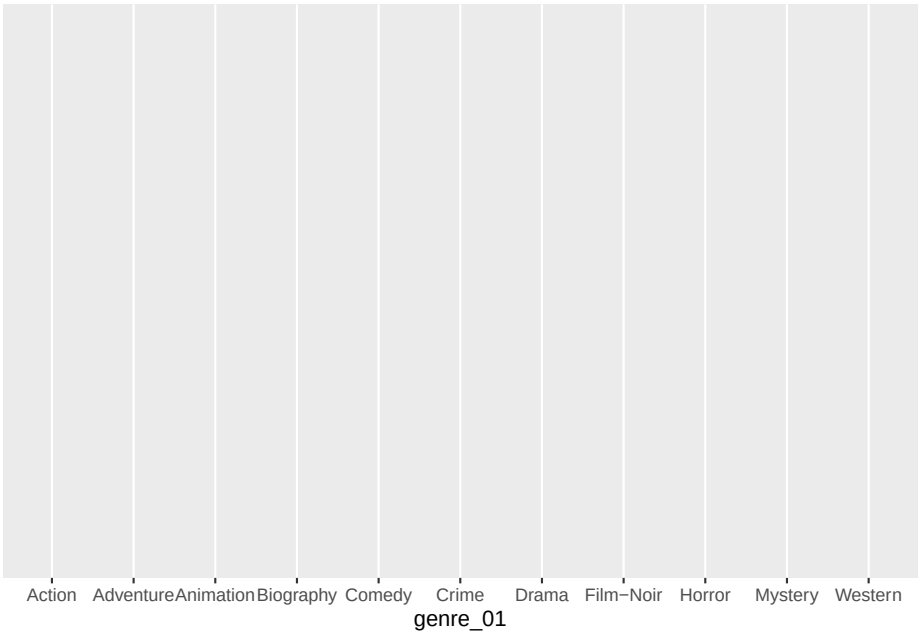
Admittedly, this is a harsh oversimplification, but it will serve as a useful guide to get us started. The function `ggplot()` is the one responsible to create any type of data visualisation. The generic structure of a `ggplot()` looks like this:

```
ggplot(data, aes(x = variable_01, y = variable_02))
```

In other words, we first need to provide the dataset `data`, and then the aesthetics (`aes()`). Think of `aes()` as the place where we define our variables (i.e. `x` and `y`). For example, we might be interested to know which movie genre is the most popular among the top 250 IMDb movies. The dataset `imdb_top_250` from the `r4np` package allows us to find an answer to this question. Therefore we define the components of the plot as follows:

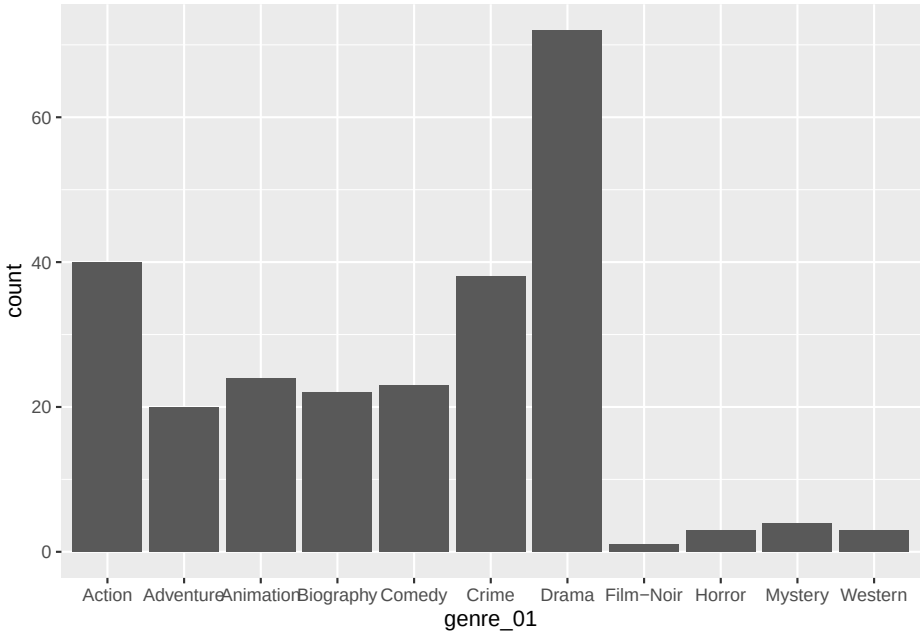
- our data is `imdb_top_250`, and
- our variable of interest is `genre_01`

```
ggplot(imdb_top_250, aes(x = genre_01))
```



Running this line of code will produce an empty plot. We only get labels for our x-axis, since we defined it already. However, we have yet to tell `ggplot()` how we want to represent the data on this canvas. Your choice for how you want to plot your data is usually informed by the type of data you want to plot and the statistics you want to represent. For example trying to plot the mean of a factor is not useful, e.g. computing the mean of movie genres. On the other hand, we can count how often certain genres appear in our dataset. One way of representing the count (or frequency) of a factor is to use a bar plot. In order to add an element to `ggplot()`, i.e. bars, we use `+` and append the function `geom_bar()` which draws bars. The `+` operator works similar to `%>%` and allows to chain multiple functions one after the other.

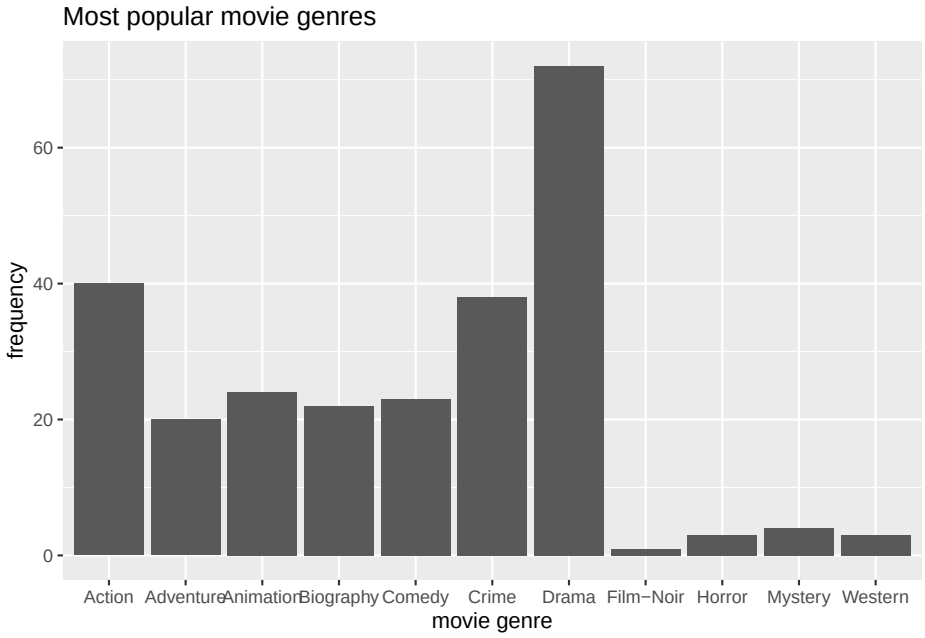
```
ggplot(imdb_top_250, aes(x = genre_01)) +
  geom_bar()
```



This already looks great and we can see that `Drama` is by far the most popular genre, followed by `Action` and `Crime`. Thus, we successfully found an answer to our question. Still, there are more improvements necessary to use it in a publication.

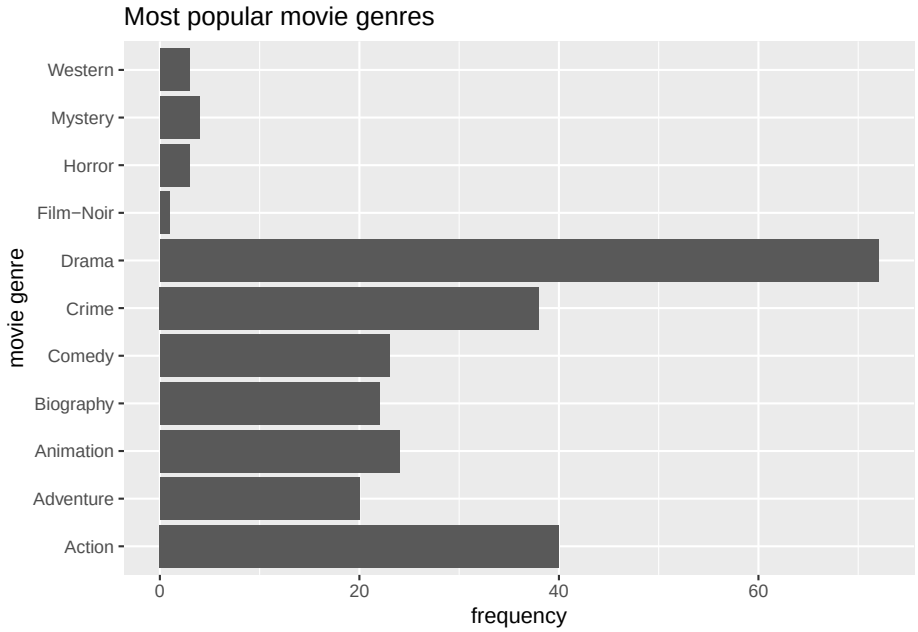
We can use `+` to add other elements to our plot, such as a title and proper axes labels. Here are some common functions to further customise our plot:

```
ggplot(imdb_top_250, aes(x = genre_01)) +
  geom_bar() +
  ggtitle("Most popular movie genres") +  # Add a title
  xlab("movie genre") +                   # Rename x-axis
  ylab("frequency")                       # Rename y-axis
```



When working with factors, the category names can be rather long. In this plot, we also have lots of categories and the labels `Adventure`, `Animation` and `Biography` are a bit too close to each other for my taste. This might be a good opportunity to use `coord_flip()`, which rotates the entire plot by 90 degrees, i.e. turning the x-axis into the y-axis and vice versa. This makes the label much easier to read.

```
ggplot(imdb_top_250, aes(x = genre_01)) +
  geom_bar() +
  ggtitle("Most popular movie genres") +
  xlab("movie genre") +
  ylab("frequency") +
  coord_flip()
```



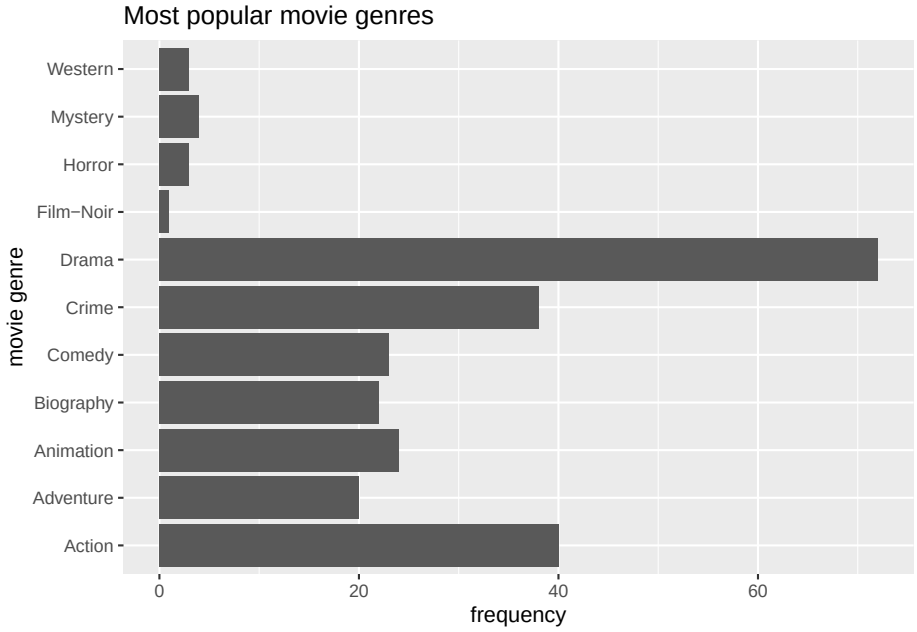
Our plot is almost perfect, but there is one more step we should take to make reading and understanding this plot even easier. At the moment, the bars are ordered alphabetically by movie genre. This is hardly ever a useful way to order your data. Instead, we might want to sort the data by the frequency, showing the most popular genre at the top. We could either sort the movies by hand or slightly amend what we have coded so far.

The problem you encounter when rearranging a `geom_bar()` with only one variable is that we do not have an explicit value we can use to indicate how we want to sort the bars. Our current code is based on the fact that `ggplot` does the counting for us. So, instead we need to do two things:

- create a table with all genres and their frequency, and
- use this table to plot the genres by the frequency we computed

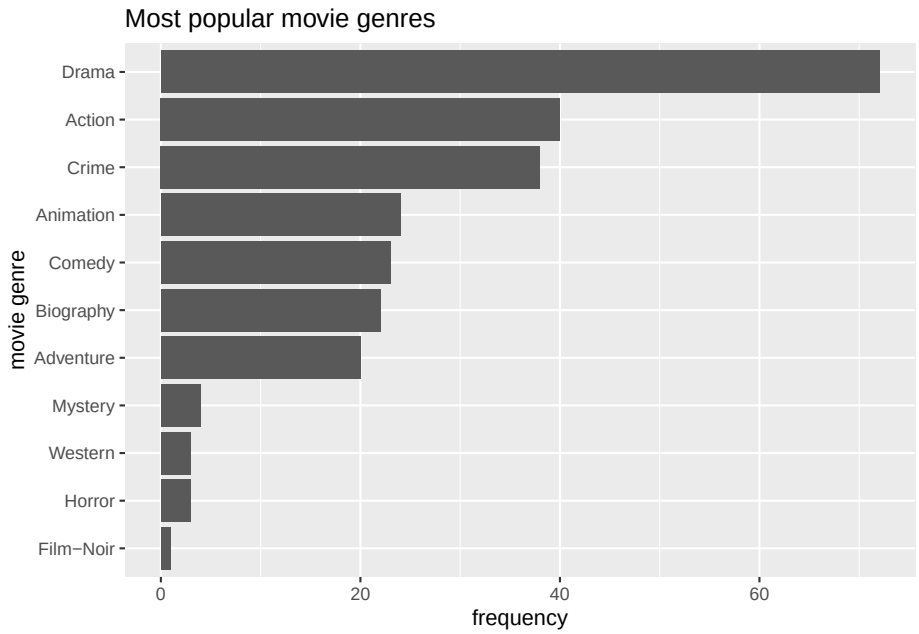
```
# Step 1: The frequency table only
imdb_top_250 %>%
  count(genre_01)
## # A tibble: 11 x 2
##    genre_01      n
##    <fct>     <int>
##  1 Action       40
##  2 Adventure    20
##  3 Animation    24
```

```
##  4 Biography    22
##  5 Comedy       23
##  6 Crime        38
##  7 Drama        72
##  8 Film-Noir     1
##  9 Horror        3
## 10 Mystery       4
## 11 Western       3

# Step 2: Plotting a barplot based on the frequency table
imdb_top_250 %>%
  count(genre_01) %>%
  ggplot(aes(x = genre_01, y = n)) +
  geom_col() +                                  # Use geom_col() instead of geom_bar()
  ggtitle("Most popular movie genres") +
  xlab("movie genre") +
  ylab("frequency") +
  coord_flip()
```




```
#Step 3: Sort the plot by frequency, i.e. by 'n'
imdb_top_250 %>%
  count(genre_01) %>%
  ggplot(aes(x = reorder(genre_01, n), y = n)) +    # Use 'reorder()'
```

```
  geom_col() +
  ggtitle("Most popular movie genres") +
  xlab("movie genre") +
  ylab("frequency") +
  coord_flip()
```



Step 3 is the only code you need to create the desired plot. The two other steps are a mean to demonstrate how one can slowly built up this plot, step-by-step. You might have noticed that I used `dplyr` to chain all these functions together and therefore it was not necessary to specify the dataset in `ggplot()`.

There are also two new functions we had to use: `geom_col()` and `reorder()`. The functions `geom_col()` and `geom_bar()` can be confusing. The easiest way to remember how to use them is: If you use a frequency table to create your barplot, use `geom_col`, if not, use `geom_bar()`. The function `geom_col()` requires that we specify a y-axis (our frequency scores), while `geom_bar()` does not.

In many cases, when creating plots you have to perform two steps:

- generate the statistics you want to plot, e.g. a frequency table, and
- plot the data via `ggplot()`

Now you have learned the fundamentals of plotting in R. Throughout the next chapters we will create a lot more plots. They all share the same basic structure, but we will use different 'geoms' to describe our data. By the end of this book, you will have accrued enough experience in plotting in R that it will feel like second nature. If you want to deepen your knowledge of `ggplot`, you should take a look at the book 'ggplot: Elegant Graphics for Data Analysis' or 'R Graphics Cookbook' which moves beyond ggplots.

8.2 Central tendency measures: Mean, Median, Mode

The mean, median and mode (the 3 Ms), are all measures of central tendency, i.e. they provide insights into how our data is distributed. All three of these measures summarise our data/variable by a single score which can be helpful, but sometimes also terribly misleading.

8.2.1 Mean

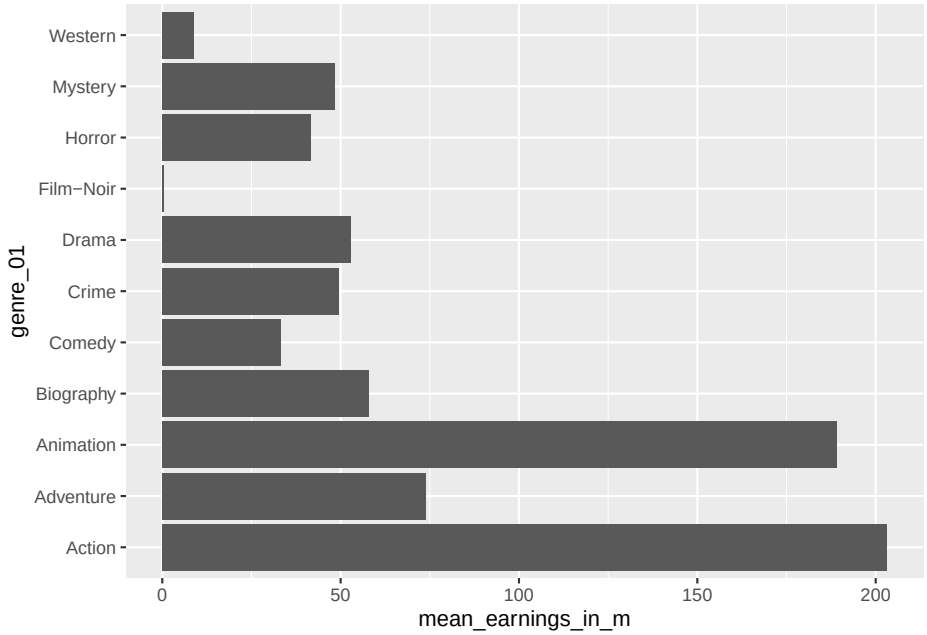
The mean is likely the most known descriptive statistics and, at the same time, a very powerful and influential one. For example, the average ratings of restaurants on Google might influence our decision on where to eat out. Similarly, we might consider the average rating of movies to decide which one to watch in cinema with friends. What we casually refer to as the 'average' is equivalent to the 'mean'.

In R it is simple to compute the mean using the function `mean()`. We used this function in Chapter 5.3 and Chapter 7.7 already. However, we have not looked at how we can plot the mean.

Assume, we are curious to know how successful movies are in each of the genres. The mean could be a good starting point to answer this question, because it provides the 'average' success of movies in a genre. The simplest approach to investigate this is to use a bar plot, like in Chapter 8.1. We first create a tibble that contains the means of `gross_in_m` for each genre in `genre_01`. Then we use this table to plot a bar plot with `geom_col()`.

```
imdb_top_250 %>%
  # Group data by genre
  group_by(genre_01) %>%
  # Compute the mean for each group (remove NAs via na.rm = TRUE)
  summarise(mean_earnings_in_m = mean(gross_in_m, na.rm = TRUE)) %>%
# Create the plot
  ggplot(aes(x = genre_01, y = mean_earnings_in_m)) +
```

```
  geom_col() +
  coord_flip()
```



It appears as if `Action` and `Animation` are far ahead of the rest. On average, both make around 200 million per movie. `Adventure` ranks third with only around 70 million. We can retrieve the exact earnings by removing the plot from the above code.

```
imdb_top_250 %>%
  group_by(genre_01) %>%
  summarise(mean_earnings_in_m = mean(gross_in_m, na.rm = TRUE)) %>%
  arrange(desc(mean_earnings_in_m))
## # A tibble: 11 x 2
##    genre_01  mean_earnings_in_m
##    <fct>                  <dbl>
##  1 Action                203.
##  2 Animation             189.
##  3 Adventure              73.9
##  4 Biography              57.9
##  5 Drama                  52.9
##  6 Crime                  49.6
##  7 Mystery                48.4
##  8 Horror                 41.6
```

```
##  9 Comedy                  33.2
## 10 Western                  8.81
## 11 Film-Noir                0.45
```

In the last line I used a new function called `arrange()`. It allows us to sort rows in our dataset by a specified variable (i.e. a column). By default, `arrange()` sorts values in ascending order, which would put the top genre last. Therefore, we have to use another function to change the order to descending with `desc()`. Now the top genre is listed at the top.

Based on this result we might believe that `Action` and `Animation` movies are the most successful genres. However, we have not taken into account how many movies there are in each genre. Consider the following example:

```
# Assume there are 2 Action movies in the top 250
2 * 203
## [1] 406

# Assume there are 10 Drama movies in the top 250
10 * 53
## [1] 530
```

Thus, the 'mean' alone might not be a sufficient indicator. It can tells us which genre is most successful based on a single movie, but we also should consider how many movies there are in each genre.

Let's add the number of movies (`n`) in each genre to our table.

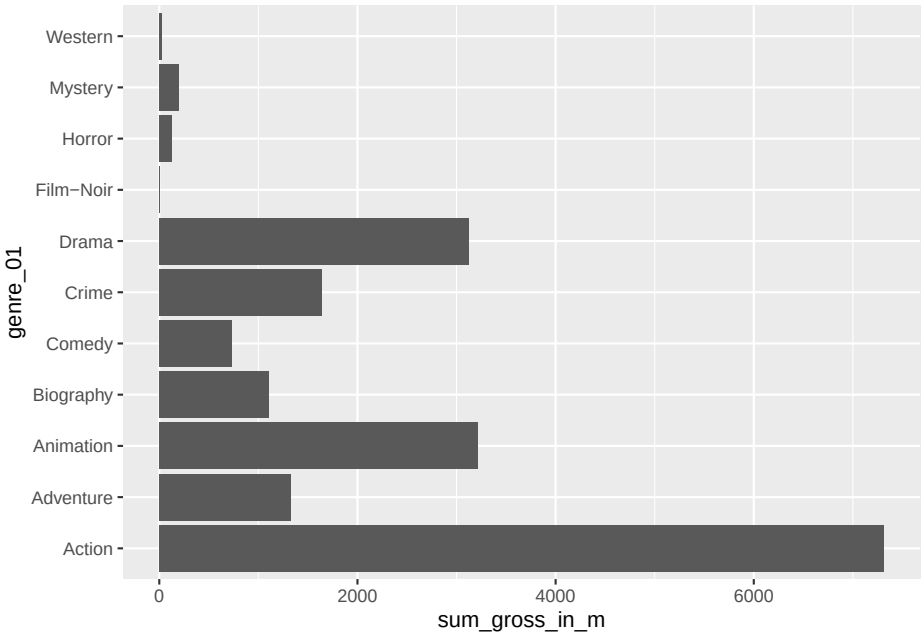
```
imdb_top_250 %>%
  group_by(genre_01) %>%
  summarise(mean_earnings_in_m = mean(gross_in_m, na.rm = TRUE),
            n = n()) %>%
  arrange(desc(mean_earnings_in_m))
## # A tibble: 11 x 3
##    genre_01  mean_earnings_in_m     n
##    <fct>                  <dbl> <int>
##  1 Action                203.      40
##  2 Animation             189.      24
##  3 Adventure              73.9     20
##  4 Biography              57.9     22
##  5 Drama                  52.9     72
##  6 Crime                  49.6     38
##  7 Mystery                48.4      4
##  8 Horror                 41.6      3
##  9 Comedy                 33.2     23
```

```
## 10 Western                8.81     3
## 11 Film-Noir              0.45     1
```

Our interpretation might slighlty change based on these findings. There are considerably more movies in the genre `Drama` then in `Action` or `Animation`. We already plotted the frequency of movies per genre in Chapter 8.1. Accordingly, we would have to think that `Drama` turns out to be the most successful genre.

As a final step, we can plot the sum of all earnings per genre as yet another indicator for the 'most successful genre'.

```
imdb_top_250 %>%
  filter(!is.na(gross_in_m)) %>%
  group_by(genre_01) %>%
  summarise(sum_gross_in_m = sum(gross_in_m)) %>%
  ggplot(aes(x = genre_01, y = sum_gross_in_m)) +
  geom_col() +
  coord_flip()
```



Theses results confirm that the `Action` genre made the most money out of all genres covered by the top 250 IMDb movies. I am sure you are curious to know which action movie contributed the most to this result. We can achieve this easily by using functions we already know.

```
imdb_top_250 %>%
  select(title, genre_01, gross_in_m) %>%
  filter(genre_01 == "Action") %>%
  arrange(desc(gross_in_m)) %>%
  top_n(5)
## Selecting by gross_in_m
## # A tibble: 5 x 3
##   title                  genre_01 gross_in_m
##   <chr>                  <fct>         <dbl>
## 1 Avengers: Endgame      Action         858.
## 2 Avengers: Infinity War Action         679.
## 3 The Dark Knight        Action         535.
## 4 The Dark Knight Rises  Action         448.
## 5 Jurassic Park          Action         402.
```

`Avengers: Endgame` and `Avengers: Infinity` war rank the highest out of all Action movies. Two amazing movies if you are into Marvel comics.

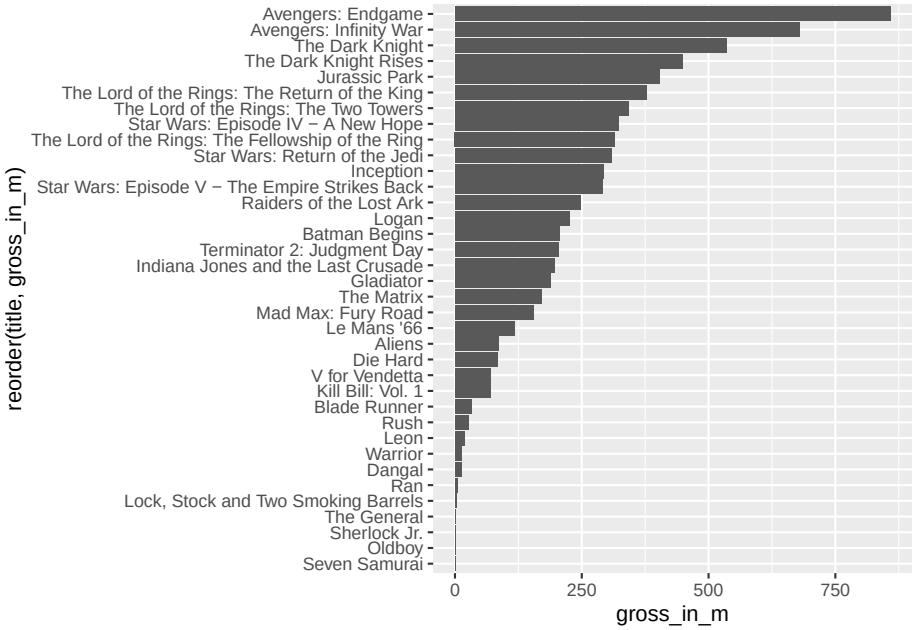
In the last line I sneaked in another new function from `dplyr` called `top_n()`. This function allows us to pick any given number of rows from the top. If you have many rows in your dataset (remember there are 40 action movies), you might want to be 'picky' and report only the top 3, 4 or 5.

In conclusion, the mean is helpful in understanding how each individual movie on average performed across different genres. However, the mean alone provides a rather incomplete picture. Thus, we need always look at means in context of other information we have to gain a more comprehensive insight into our data.

8.2.2 Median

The 'median' is the little, lesser known and used brother of the 'mean'. However, it can be a very powerful indicator for central tendency, because it is not so much affected by outliers. With outliers I mean observations that lie way beyond or below the average observation in our dataset. Let's inspect the `Action` genre more closely and see how each movie in this category performed relative to each other. We first `filter()` our dataset to only show movies in the genre `Action` and also remove responses that have no value for `gross_in_m`.

```
imdb_top_250 %>%
  filter(genre_01 == "Action" & !is.na(gross_in_m)) %>%
  ggplot(aes(x = reorder(title, gross_in_m), y = gross_in_m)) +
  geom_col() +
  coord_flip()
```



`Avengers: Endgame` and `Avengers: Infinity War` are far ahead of any other movie. We can compute the mean with and without these two movies.

```
# Mean earnings for Action genre with Avengers movies
imdb_top_250 %>%
  filter(genre_01 == "Action") %>%
  summarise(mean = mean(gross_in_m, na.rm = TRUE))
## # A tibble: 1 x 1
##    mean
##   <dbl>
## 1  203.

# Mean earnings for Action genre with Avengers movies
imdb_top_250 %>%
  filter(genre_01 == "Action" &
           title != "Avengers: Endgame" &
           title != "Avengers: Infinity War") %>%
  summarise(mean = mean(gross_in_m, na.rm = TRUE))
## # A tibble: 1 x 1
##    mean
##   <dbl>
## 1  170.
```

The result is striking. Without these two movies, the `Action` genre would have

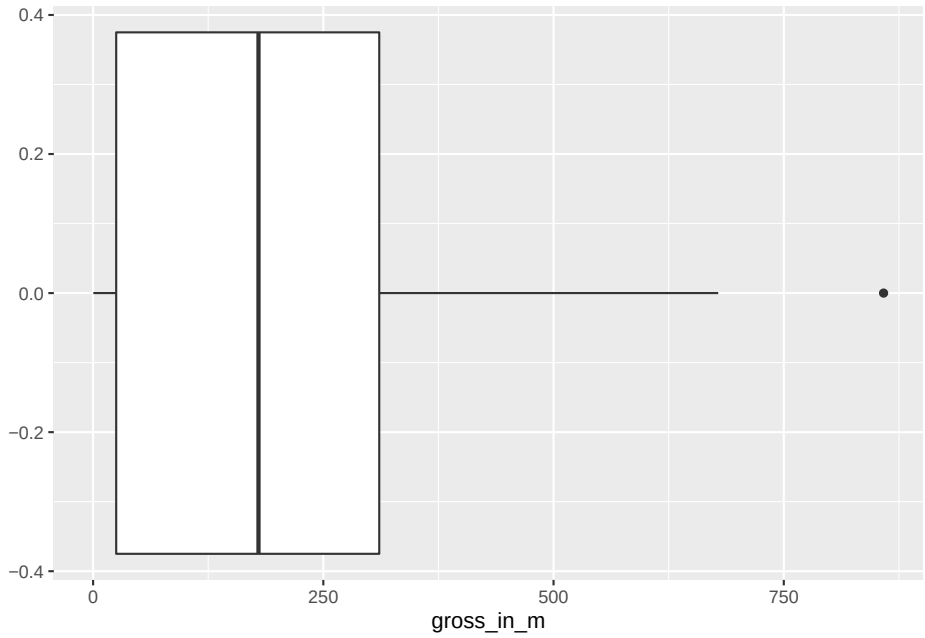
less earning per movie than the `Animiation` genre. However, if we computed the median instead, we would not notice such a huge difference in results. This is due to the fact that the median sorts a dataset, e.g. by `gross_in_m` and then picks the value that would cut the data into two equally large halves. It does not matter which value is the highest or lowest in our dataset, what matters is the value that is ranked right in the middle of all values.

```
# Median earnings for Action genre with Avengers movies
imdb_top_250 %>%
  filter(genre_01 == "Action") %>%
  summarise(median = median(gross_in_m, na.rm = TRUE))
## # A tibble: 1 x 1
##   median
##    <dbl>
## 1   180.

# Median earnings for Action genre with Avengers movies
imdb_top_250 %>%
  filter(genre_01 == "Action" &
           title != "Avengers: Endgame" &
           title != "Avengers: Infinity War") %>%
  summarise(median = median(gross_in_m, na.rm = TRUE))
## # A tibble: 1 x 1
##   median
##    <dbl>
## 1   163.
```

Both medians are much closer to each other, showing how much better suited the median is in our case. In general, when we report means, it is advisable to report the median as well. If a mean and median differ substantially, it could mean your data 'suffers' from outliers. So we have to detect them and think about whether we should remove them for further analysis.

We can visualise medians using boxplots. Boxplots are a very effective tool to show how your data is distributed in one single data visualisation and it shows more than just the mean. It also shows the spread of your data (see Chapter @ref()).

```
imdb_top_250 %>%
  filter(genre_01 == "Action" & !is.na(gross_in_m)) %>%
  ggplot(aes(gross_in_m)) +
  geom_boxplot()
```



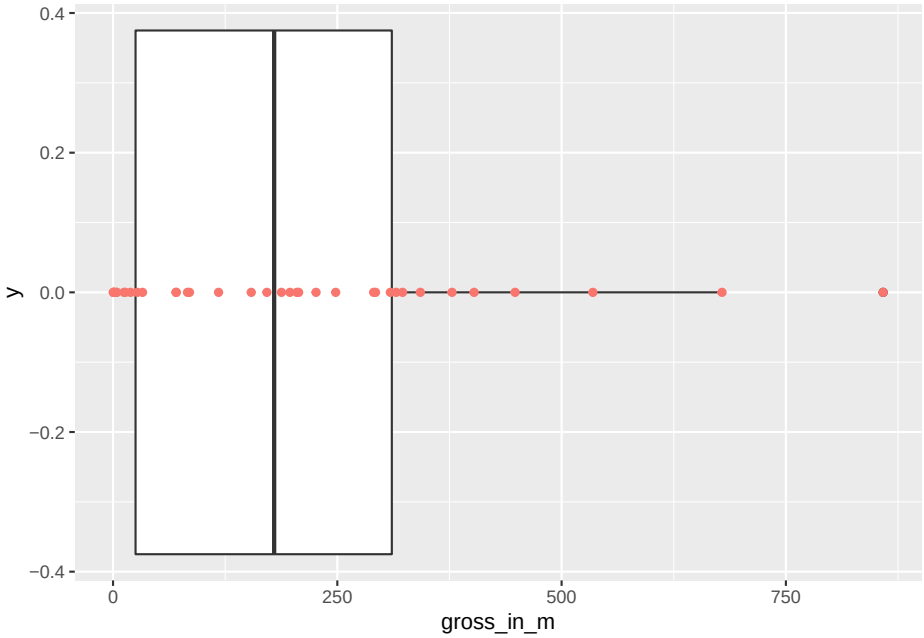
To interpret this boxplot consider the Figure **??**.

`[insert boxplot explanation figure here]`

The boxplot shows that we have apparently one observation that is an outlier, which is likely `Avengers: Endgame`, but what happened to the over Avengers movie? Is it no an outlier too? It seems we need some more information. We can overlay another `geom_` on top to visualise where exactly each movie lies on this boxplot. We can represent each movie as a point by using `geom_point()`.

This function requires us to define the values for the x and y axis. Here it makes sense to set `y = 0` which aligns all the dots in the middle of the boxplot.

```
imdb_top_250 %>%
  filter(genre_01 == "Action" & !is.na(gross_in_m)) %>%
  ggplot(aes(gross_in_m)) +
  geom_boxplot() +
  geom_point(aes(y = 0, colour = "red"), show.legend = FALSE)
```



To make the dots stand out more I changed the colour to `"red"`. By adding the `colour` (`color` also works) attribute ggplot would automatically generate a legend. Since we do not need it we can specify it directly in the `geom_point()` function. If you prefer the legend, just remove `show.legend = FALSE`.

The two dots on the right are the two Avengers movies. This plot also nicely demonstrates why boxplots are so popular and helpful: They provide so many insights, not only into the central tendency of a variable, but also highlights outliers and, more generally, gives a sense of the spread of our data (more about this in Chapter @ref()).

The median is an important descriptive and diagnostic statistic and should be included in most empirical quantitative studies.

8.2.3 Mode

Finally, the 'mode', indicates which value is the most frequently occurring value for a certain variable. For example, we might be interested in knowing which IMDb rating was most frequently awarded to the top 250 movies.

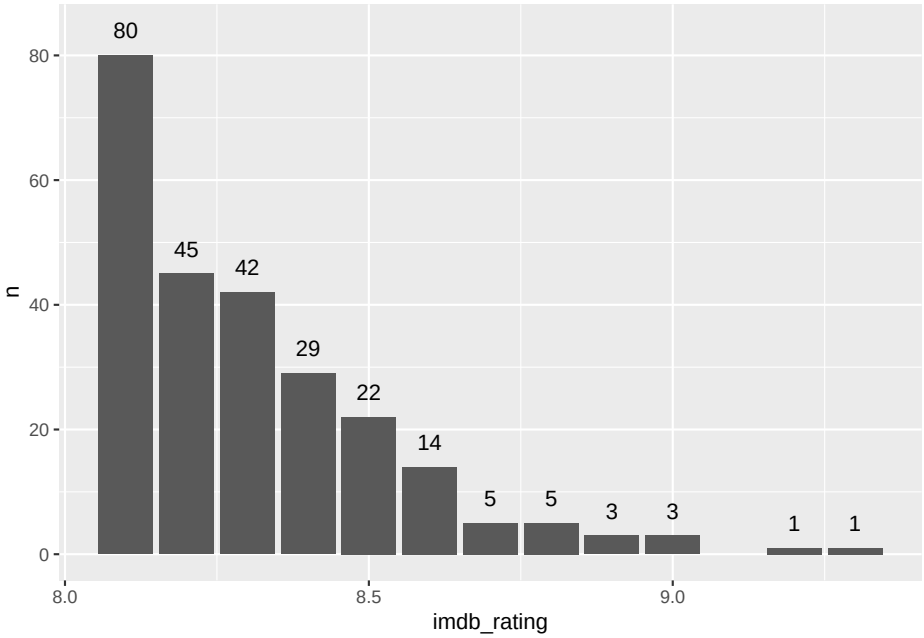
When trying to compute the mode in R, we quickly run into a problem, because there is no function available to do this straight away unless you search for a package that does it for you. However, before you start searching, let's reflect on what the mode does and why we can find out the mode without additional packages or functions. The mode is based on the frequency of the

occurrence of a value. Thus, the most frequently occurring value would be the one that is listed at the top of a frequency table. We have already created several frequency tables in this book and we can create another one to find the answer to our question.

```
imdb_top_250 %>% count(imdb_rating)
## # A tibble: 12 x 2
##    imdb_rating     n
##          <dbl> <int>
##  1         8.1    80
##  2         8.2    45
##  3         8.3    42
##  4         8.4    29
##  5         8.5    22
##  6         8.6    14
##  7         8.7     5
##  8         8.8     5
##  9         8.9     3
## 10         9       3
## 11         9.2     1
## 12         9.3     1
```

We can also easily visualise this frequency table in the same way as before. In order to make the plot a bit more 'fancy', we can add labels to the bar which reflect the frequency of the rating. We need to add the attribute `label` and also add a `geom_text()` layer to display them. Because the numbers would overlap with the bars, I 'nudge' the labels up by `4` units on the y axis.

```
imdb_top_250 %>%
  count(imdb_rating) %>%
  ggplot(aes(imdb_rating, n, label = n)) +
  geom_col() +
  geom_text(nudge_y = 4)
```



The frequency table and the plot reveal that the mode for `imdb_rating` is `8.1`. In addition, we also get to know how often this rating was applied, i.e. `80` times. This provides much more information then receiving a single score and helps to better interpret the importance of the mode as an indicator for central tendency. Consequently, there is generally no need to compute the mode if you can have a frequency table instead. Still, if you are keen to have a function that computes the mode, you will have to write your own function, e.g. as shown in this post on stackeroverflow.com or search for a package that coded one already.

As a final remark, it is also possible that you can find two or more modes in your data. For example, if the rating `8.1` appears `80` times and the rating `9.0` appears `80` times, both would be considered to a mode.

8.3 Indicators and visualisations to examine the spread of data

Understanding how your data is spread out is important to get a better sense of what your data is composed of. We already touched upon the notion of spread in Chapter 8.2.2 through plotting a boxplot. The spread of data provides insights into how homogeneous or heterogeneous the responses of our participants are. In the following we will cover some essential techniques to investigate the spread of your data and investigate whether our variables are

normally distributed, which is often an important assumption for certain analytical techniques. In addition, we will aim to identify outliers which could be detrimental to subsequent analysis and significantly affect our modelling and testing in later stages.

8.3.0.1 Boxplot: So much information in just one box

The boxplot is a staple in visualising descriptive statistics. If offers so much information in just a single plot that it might not take much to convince you that it has become a very popular way to show the spread of data.

For example, we might be interested to know how long most movies run. Our gut feeling might tell us that most movies are probably around 2 hours long. One approach to find out is a boxplot, which we used before.

```
# Text for annotations
longest_movie <- imdb_top_250 %>%
  filter(runtime_min == max(runtime_min)) %>%
  select(title)

shortest_movie <- imdb_top_250 %>%
  filter(runtime_min == min(runtime_min)) %>%
  select(title)

# Create the pot
imdb_top_250 %>%
  ggplot(aes(runtime_min)) +
  geom_boxplot() +
  annotate("text",
           label = glue::glue("{longest_movie}
                              ({max(imdb_top_250$runtime_min)} min)"),
           x = 310,
           y = 0.05,
           size = 2.5) +
  annotate("text",
           label = glue::glue("{shortest_movie}
                              ({min(imdb_top_250$runtime_min)} min)"),
           x = 45,
           y = 0.05,
           size = 2.5)
```

The results indicate that most movies are between 100 to 150 minutes long. Our intuition was right. We find that one movie is even over 300 minutes long, i.e. over 5 hours: 'Gangs of Wasseypur'. In contrast, the shortest movie only lasts 45 minutes and is called 'Sherlock Jr.'. I added annotations using

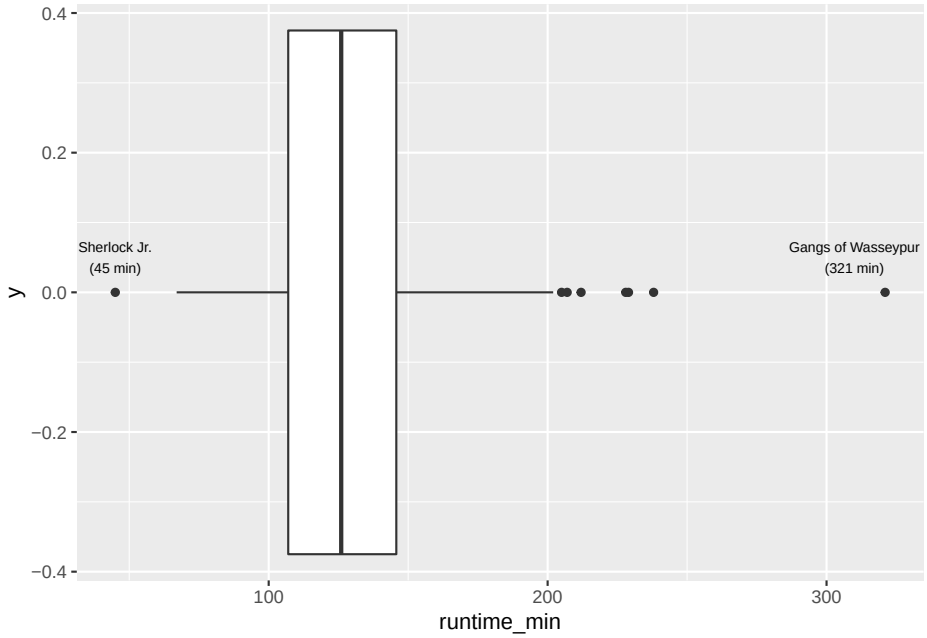


Figure 8.1: A boxplot

`annotate()` to highlight these two movies in the plot. A very useful package for annotations is `glue`, which allows to combine text with data. So, instead of looking up the titles of the longest and shortest movie, I used the `filter()` and `select()` functions to find them automatically. This has the great advantage that if I wanted to update my data, the name of the longest movie might change. However, I do not have to look it up again by hand.

As we can see from our visualisation, both movies would also count as outliers in our dataset (see Chapter 8.3.3.2.

8.3.0.2 Histogram: Do not mistake it as a bar plot

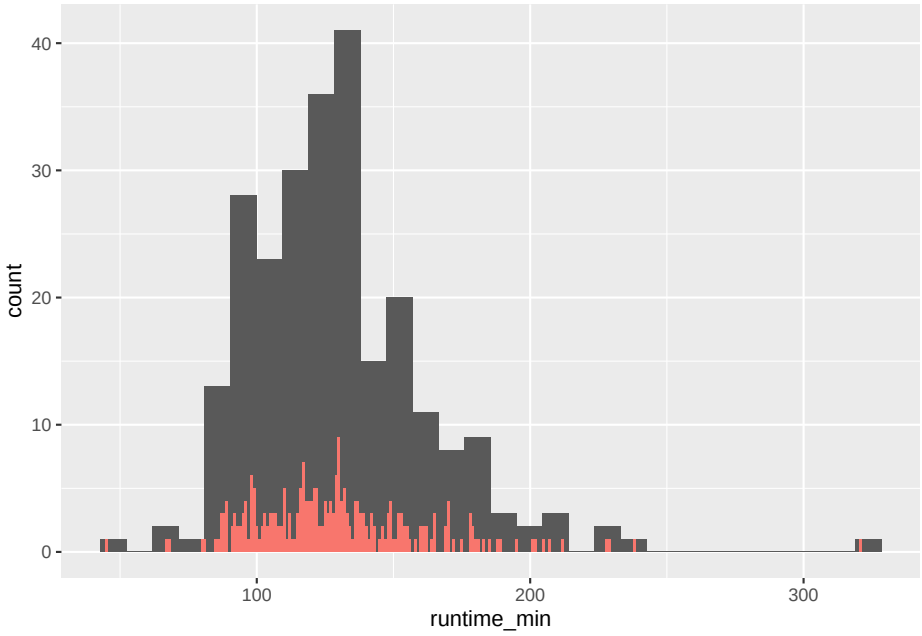
Another frequently used approach to show the spread (or distribution) of data is the histogram. The histogram easily gets confused with a bar plot. However, you would be very mistaken to assume that they are the same. Some key differences between these two types of plots is summarised in Table 8.1.

Table 8.1: Histogram vs bar plot

Histogram	Bar plot
Used to show the distribution of non-categorical data	Used for showing the frequency of categorical data, i.e. `factors`
Each bar (also called 'bin') represents a group of observations.	Each bar represents one category (or level) in our `factor`.
The order of the bars is important and cannot/should not be changed.	The order of bars is arbitrary and can be reordered if meaningful.

To make this difference even clearer, let's overlap a bar plot with a histogram for the same variable.

```
imdb_top_250 %>%
  ggplot(aes(runtime_min)) +
  geom_histogram() +
  geom_bar(aes(fill = "red"), show.legend = FALSE)
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



There are a couple of important observations to be made:

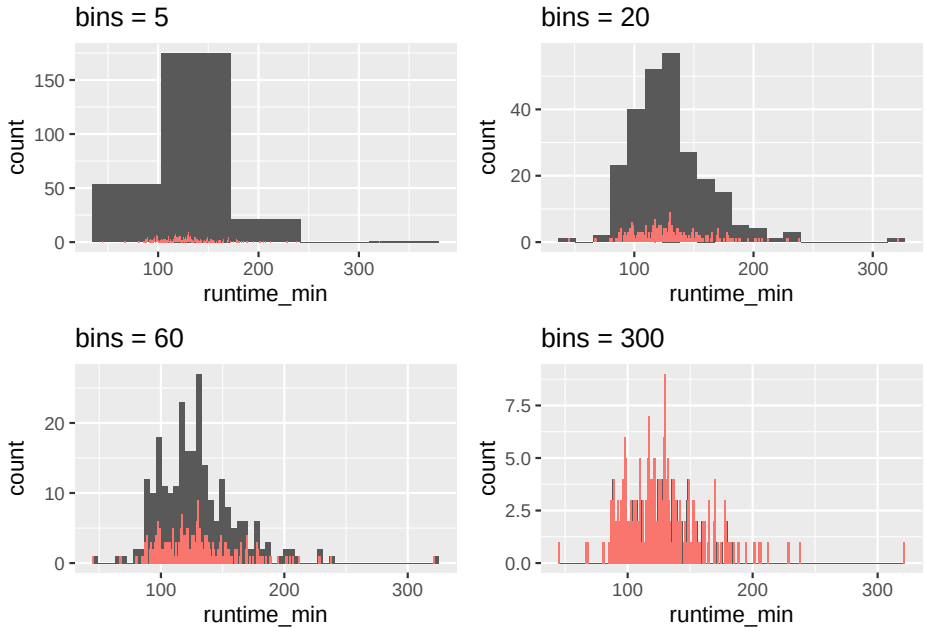
- The bar plot has much shorter bars, because each bar represents the frequency of a single unique score. Thus, the runtime of 151 is represented

as a bar as is the runtime of 152. Only identical observations are grouped together. As such, the bar plot is based on a frequency table, similar to what we computed before.

- In contrast, the histogram's 'bars' are higher because they group together individual observations based on a specified range, e.g. one bar might represent movies that have a runtime between 150-170 minutes. These ranges are commonly called `bins`.

We can control the number of bars in our histogram using the `bins` attribute. `ggplot` even reminds us in a warning that we should adjust it to represent more details in the plot. Let's experiment with this setting to see how it would look like with different numbers of `bins`.

```
imdb_top_250 %>%
  ggplot(aes(runtime_min)) +
  geom_histogram(bins = 5) +
  geom_bar(aes(fill = "red"), show.legend = FALSE)

imdb_top_250 %>%
  ggplot(aes(runtime_min)) +
  geom_histogram(bins = 20) +
  geom_bar(aes(fill = "red"), show.legend = FALSE)

imdb_top_250 %>%
  ggplot(aes(runtime_min)) +
  geom_histogram(bins = 60) +
  geom_bar(aes(fill = "red"), show.legend = FALSE)

imdb_top_250 %>%
  ggplot(aes(runtime_min)) +
  geom_histogram(bins = 300) +
  geom_bar(aes(fill = "red"), show.legend = FALSE)
```



As becomes obvious, if we select a large enough number of bins, we can achieve the same result as a bar plot. This is the closest a bar plot can become to a histogram, i.e. if you define the number of `bins` in such a way that each observation is captured by one `bin`. While theoretically possible, practically this rarely makes much sense.

We use histograms to judge whether our data is normally distributed. A normal distribution is often a requirement for assessing whether we can run certain types of analyses or not. In short: It is very important to know about it in advance. The shape of a normal distribution looks like a bell (see Figure 8.2). If our data is equal to a normal distribution we find that

- the mean and the median are the same value, and can conclude that
- the mean is a good representation for our data/variable.

Let's see whether our data is normally distributed using the histogram we already plotted and overlay a normal distribution. The coding for the normal distribution is a little more advanced. Do not worry if you cannot fully decipher its meaning just yet. To draw such a reference plot we need to:

- compute the `mean()` of our variable,
- compute the standard deviation `sd()` of our variable (see Chapter 8.3.2,
- use the function `geom_func()` to plot it and,

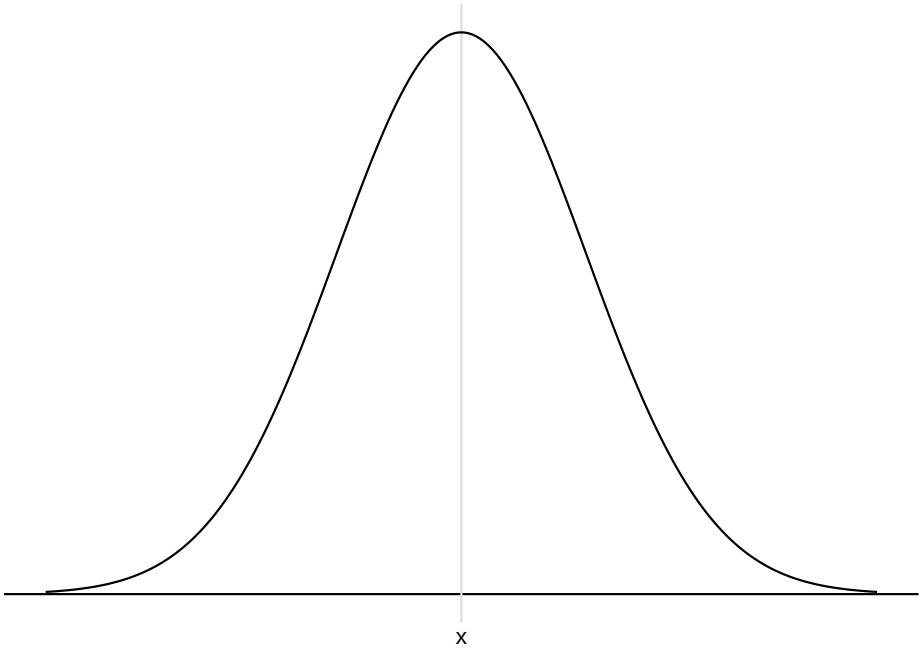
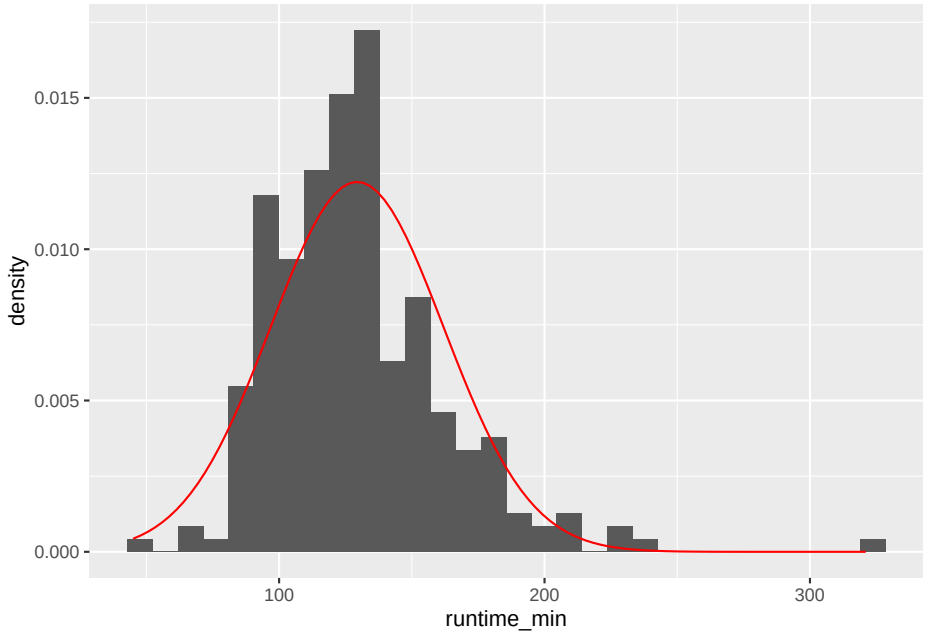


Figure 8.2: A normal distribution

- define the function `fun` as `dnorm`, which stands for 'normal distribution'.

More important than understanding how the computational side works, is the aim of this task: we try to compare our distribution with a normal one.

```
imdb_top_250 %>%
  ggplot(aes(runtime_min)) +
  geom_histogram(aes(y = ..density..), bins = 30) +

  # This part creates a normal curve based on the mean
  # and standard deviation of our data

  geom_function(fun = dnorm,
                n = 103,
                args = list(mean = mean(imdb_top_250$runtime_min),
                            sd = sd(imdb_top_250$runtime_min)),
                colour = "red")
```



However, there are two problems if we use histograms in combination with a normal distribution reference plot: First, the y axis needs to be transformed to fit the normal distribution (which is a density plot and not a histogram). Second, the shape of our histogram is affected by the number of `bins` we have chosen, which is an arbitrary choice we make. Besides, the lines of code might be tough to understand, because we have to 'hack' the visualisation to make it work. There is, however, a better way to do this: Density plots.

8.3.0.3 Density plots: Your smooth histograms

Density plots are a special form of the histogram. It uses 'kernel smoothing', which turns our blocks into a smoother shape. Better than trying to explain what it does, it might help to see it. We use the same data but replace `geom_histogram()` with `geom_density()`.

```
# Ingredients for our normality reference plot
mean_ref <- mean(imdb_top_250$runtime_min)
sd_ref <- sd(imdb_top_250$runtime_min)

imdb_top_250 %>%
  ggplot(aes(runtime_min)) +
  geom_density() +
  geom_function(fun = dnorm,
```

```
                n = 103,
                args = list(mean = mean_ref,
                            sd = sd_ref),
                colour = "red")
```

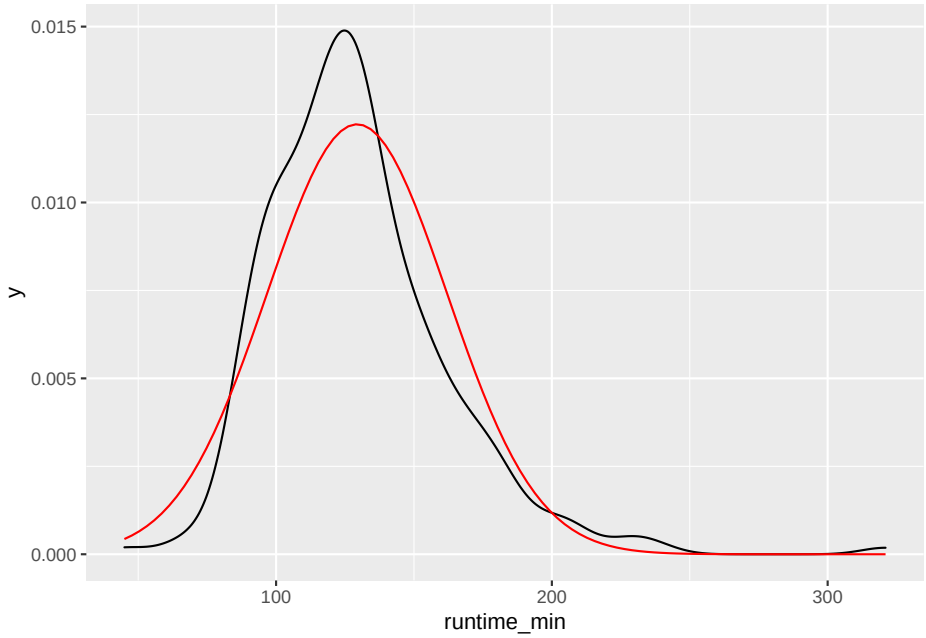
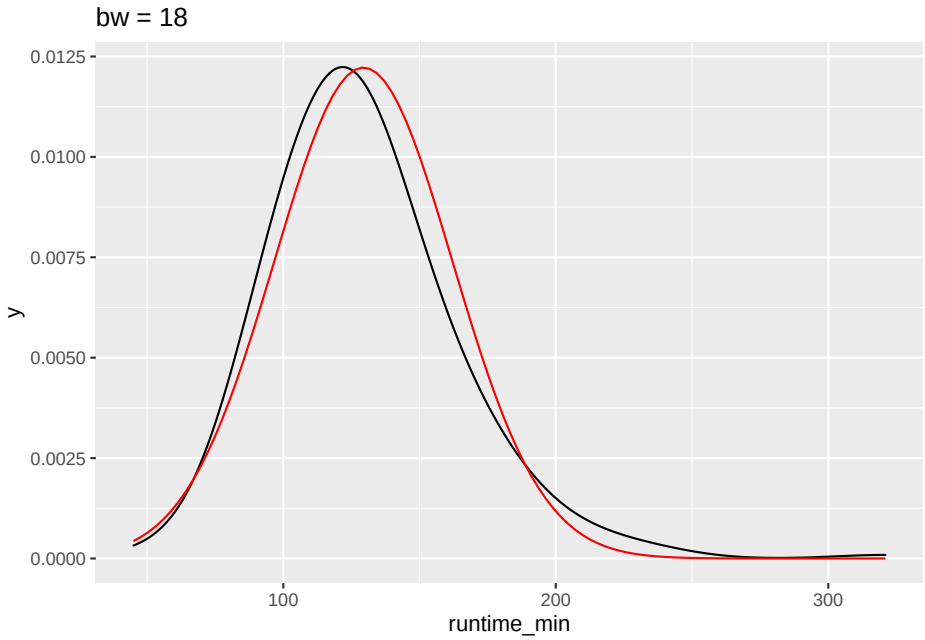


Figure 8.3: Density plot vs normal distribution

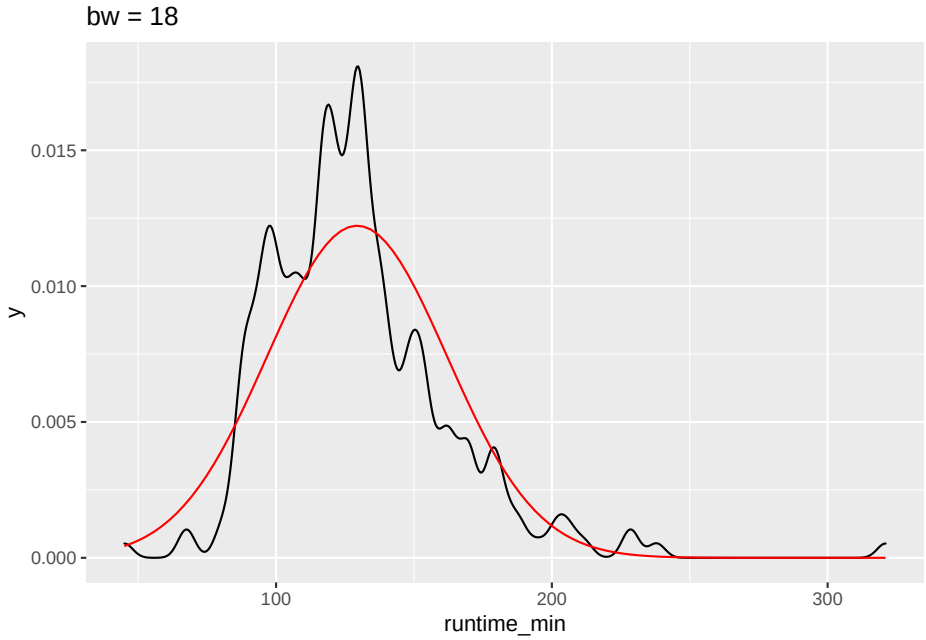
For density plots we can aslo define how big the `bins` are which make the plot more or less smooth. After all, the density plot is a histogram, but the transitions from one bin to the next are 'smoothed'. Here is an example of how different `bw` settings affect the plot.

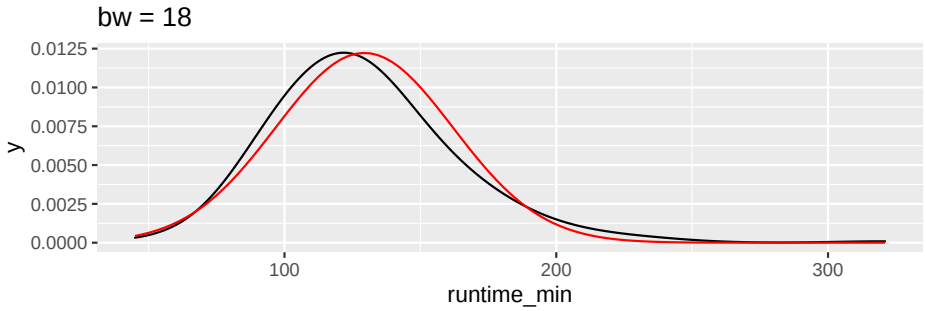
```
# Ingredients for our normality reference plot
mean_ref <- mean(imdb_top_250$runtime_min)
sd_ref <- sd(imdb_top_250$runtime_min)

# bw = 18
imdb_top_250 %>%
  ggplot(aes(runtime_min)) +
  geom_density(bw=18) +
  geom_function(fun = dnorm,
                n = 103,
                args = list(mean = mean_ref,
                            sd = sd_ref),
```

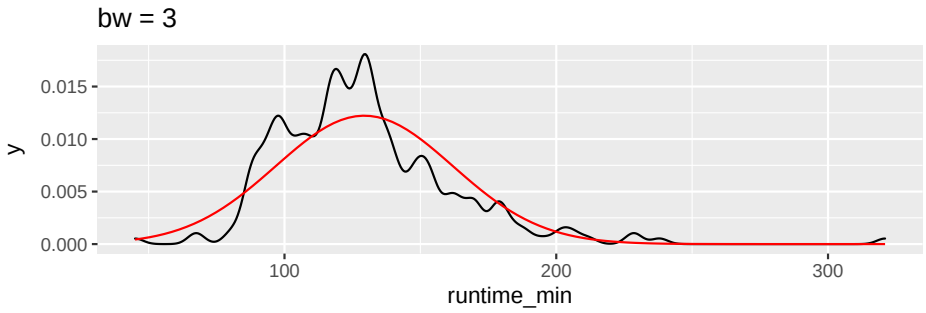
```
                colour = "red") +
  ggtitle("bw = 18")
```




```
# bw = 3
imdb_top_250 %>%
  ggplot(aes(runtime_min)) +
  geom_density(bw=3) +
  geom_function(fun = dnorm,
                n = 103,
                args = list(mean = mean_ref,
                            sd = sd_ref),
                colour = "red") +
    ggtitle("bw = 18")
```







The benefits of the density plot in this situation are obvious: It is much easier to see whether our data is normally distributed or not when compared to a

reference plot. However, we still might struggle to determine normality just by these plots, because it depends on how high or low we set `bw`.

Luckily, there is also a statistical method to test whether our data is normally distributed: The Shapiro-Wilk test. This test compares our distribution with a normal distribution (like in our plot) and tells us whether our distribution is **significantly different** from it. Thus, if the test is not significant the distribution of our data is not significantly different from a normal distribution, or in simple terms: It is normally distributed. We can run the test in R as follows:

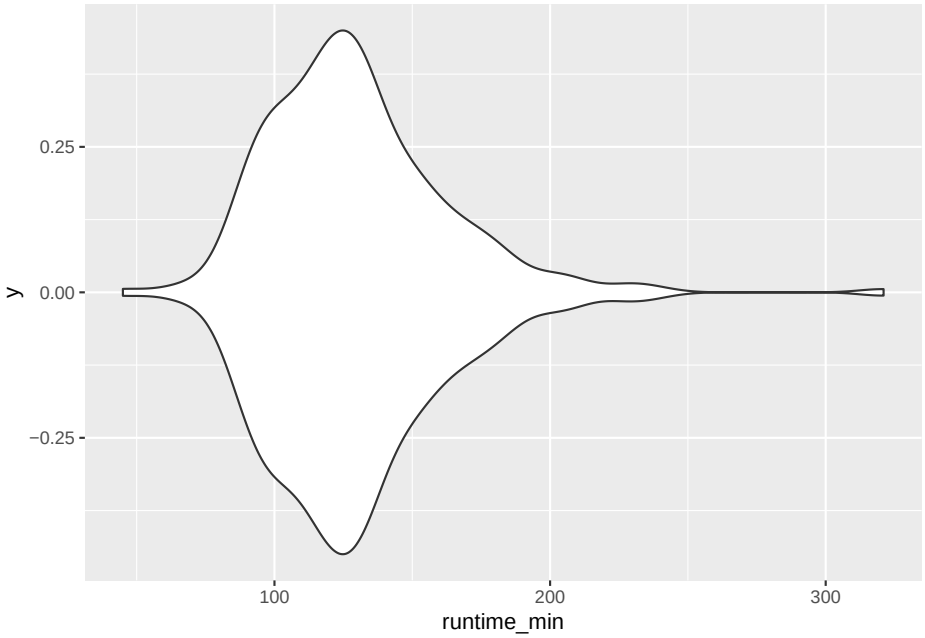
```
shapiro.test(imdb_top_250$runtime_min)
##
##  Shapiro-Wilk normality test
##
## data:  imdb_top_250$runtime_min
## W = 0.92436, p-value = 5.544e-10
```

According to this result, we have to accept that our data is not normally distributed, because it is significantly different from it ($p<0.05$). We look at significance and its meaning more thoroughly in the next chapter (Chapter 9).

8.3.0.4 Violin plot: Your smooth boxplot

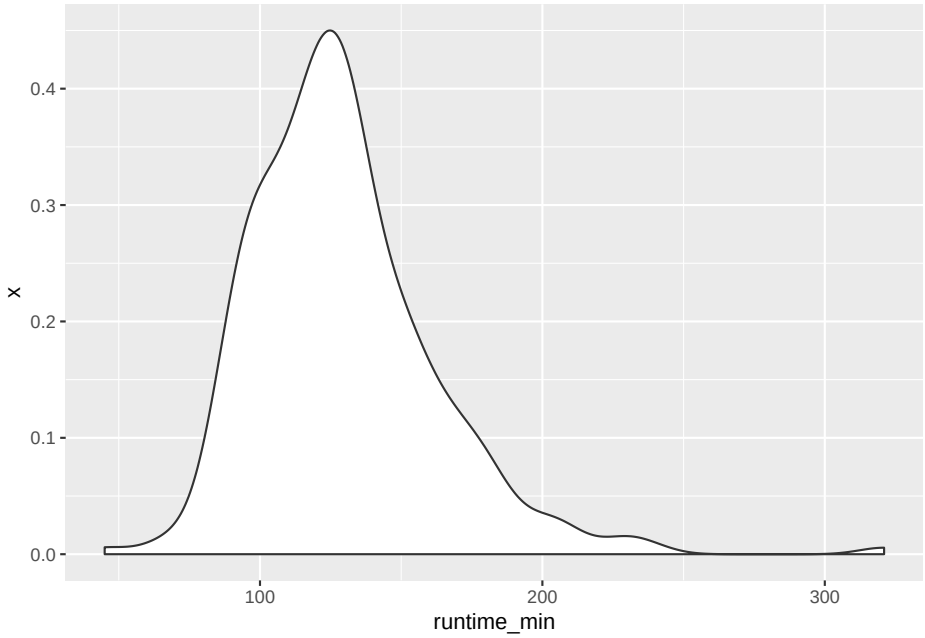
If a density plot is the sibling of a histogram, the violin plot would be the sibling of a boxplot, but the twin of a density plot. Confused? If so, then let's use the function `geom_volin()` to create one.

```
imdb_top_250 %>%
  ggplot(aes(x = runtime_min, y = 0)) +
  geom_violin()
```



Looking at our plot it becomes evident where the violin plot got its name from, i.e. its shape. The reason why it also a twin of the density plot becomes clear when we only plot half of the violin.

```
imdb_top_250 %>%
  ggplot(aes(x = 0, y = runtime_min)) +
  see::geom_violinhalf()+
  coord_flip()
```

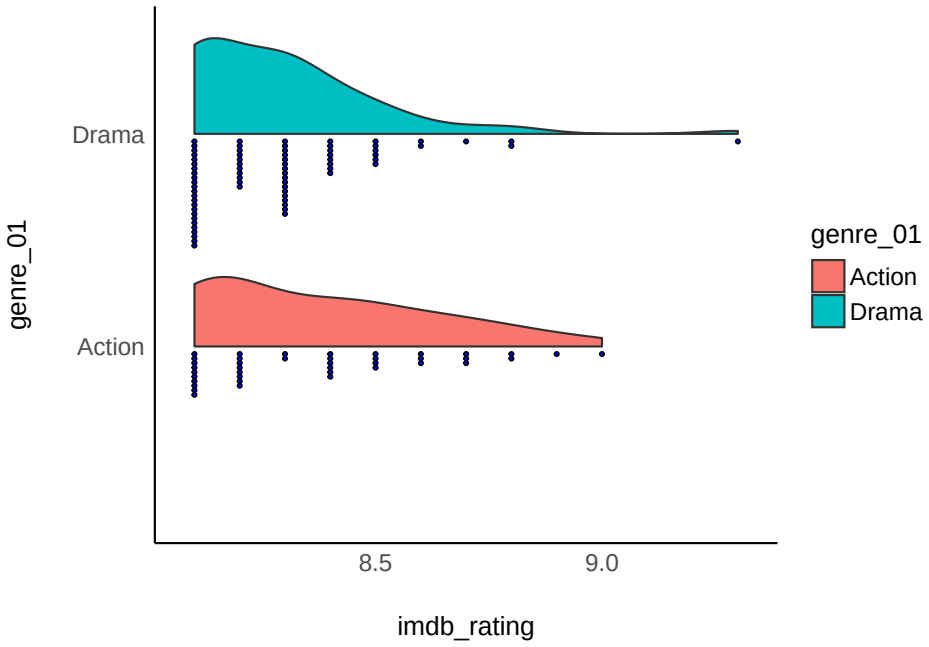


It looks exactly like the density plot we plotted earlier (see Figure 8.3. The interpretation largely remains the same to a density plot as well. The relationship between the boxplot and the violin plot lies in the symmetry of the violin plot.

At this point, it is fair to say that we enter 'fashion' territory. It is really up to your taste which visualisation you prefer, because they are largely similar, but offer nuances that some data sets might require.

Lastly, I cannot finish this chapter without sharing with you one of the most popular uses of half-violin plots: The rain cloud plot. It is a combination of a dot plot with a density plot. Each dot represents a movie in our dataset. This creates the appearance of a cloud with rain drops. There are several packages available that can create such a plot. Here I used the `see` package.

```
imdb_top_250 %>%
  filter(genre_01 == "Action" | genre_01 == "Drama") %>%
  ggplot(aes(x = genre_01, y = imdb_rating, fill = genre_01)) +
  see::geom_violindot(fill_dots = "blue", size_dots = 0.2) +
  see::theme_modern() +
  coord_flip()
```



8.3.1 QQ plot: A 'cute' plot to check for normality in your data

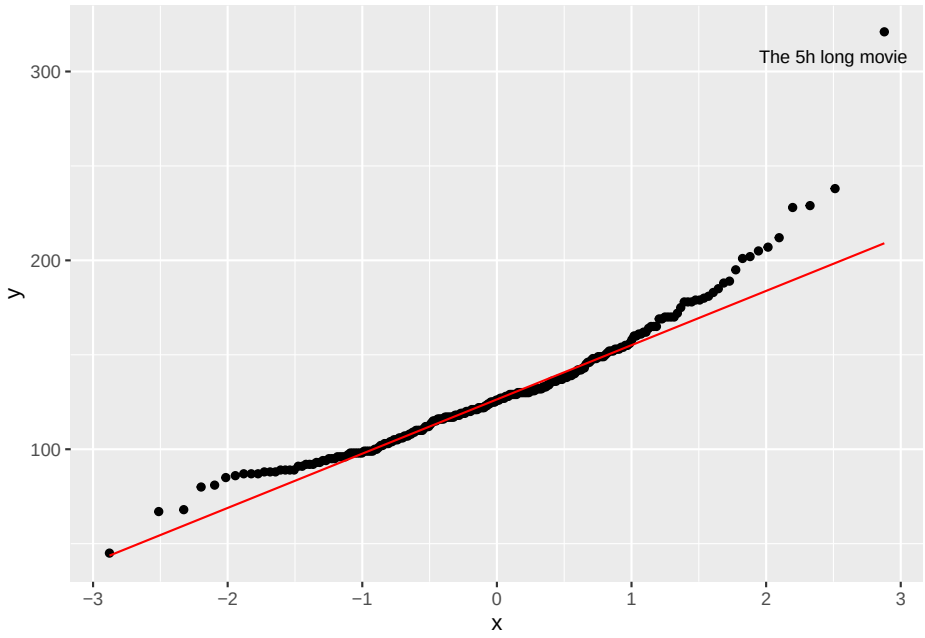
The QQ plot is an alternative to comparing distributions to a normality curve. Instead of a curve, we plot a line which represents the quantiles of our data against the quantiles of a normal distribution. The term 'quantile' can be somewhat confusing, especially after learning about the boxplot, which shows quartiles. There is a relationship between these terms. Consider the following comparison:

- 1^{st} Quartile = 25^{th} percentile = 0.25 quantile
- 2^{nd} Quartile = 50^{th} percentile = 0.50 quantile = median
- 3^{rd} Quartile = 75^{th} percentile = 0.75 quantile

With these definitions out of the way, let's plot some quantiles against each other.

```
imdb_top_250 %>%
  ggplot(aes(sample = runtime_min)) +
  geom_qq() +
  geom_qq_line(colour = "red")+
```

```
  annotate("text",
           label = "The 5h long movie",
           x = 2.5,
           y = 308,
           size = 3)
```



The function `geom_qq()` is responsible for creating the dots, while `geom_qq_line` creates a reference for a normal distribution. The reference line is drawn in such a way that it touches the quartiles of our distribution. Ideally, we would want that all dotes are firmly aligned with each other. Unfortunately, this is not the case in our dataset. At the top and at the bottom we have points that deviate quite far from a normal distribution. Remember the five hour long movie? It is very far away from the rest of the other movies in our dataset.

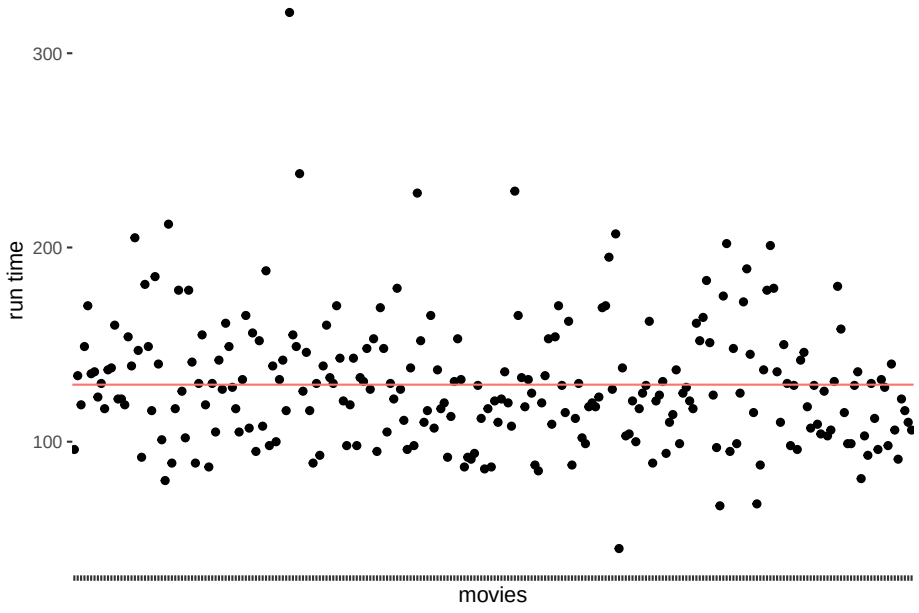
8.3.2 Standard deviation: Your average deviation from the mean

I left the most commonly reported statistics for the spread of data last. The main reasons for this lies in the fact that one easily jumps ahead to look at the standard deviation without ever considering plotting the distribution in the first place. Similar to the mean and other numeric indicators, they could

potentially convey the wrong impression. Nevertheless, the standard deviation is an import measure.

To understand what the standard deviation is, we can consider the following visualisation:

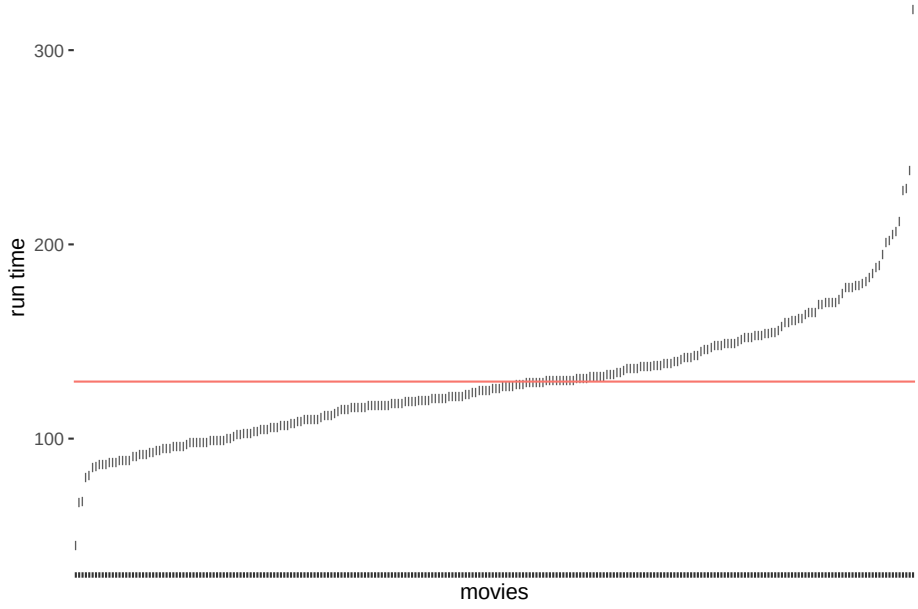
```
runtime_mean <- mean(imdb_top_250$runtime_min)

imdb_top_250 %>%
  select(title, runtime_min) %>%
  ggplot(aes(x = title, y = runtime_min)) +
  geom_point() +
  geom_hline(aes(yintercept = runtime_mean, colour = "red"), show.legend = FALSE) +

  # Making the plot a bit more pretty
  theme(axis.text.x = element_blank(),          # Removes movie titles
        panel.grid.major = element_blank(),     # Removes grid lines
        panel.background = element_blank()      # Turns background white
        ) +
  ylab("run time") +
  xlab("movies")
```



The red line represent the mean runtime for all movies in the dataset, which is 129 minutes. We notice that the points are falling around the mean, but not really directly on it. In other words, there are not many movies that are about

129 minutes long. We make this visualisation even more meaningful if we sorted the movies by their runtime. We can also change the shape of the dots (see also Chapter 9 by using the attribute `shape`. This helps to plot many dots without having them overlap.

```
imdb_top_250 %>%
  select(title, runtime_min) %>%
  ggplot(aes(x = reorder(title, runtime_min), y = runtime_min)) +
  geom_point(shape = 124) +
  geom_hline(aes(yintercept = runtime_mean, colour = "red"), show.legend = FALSE) +

  theme(axis.text.x = element_blank(),
        panel.grid.major = element_blank(),
        panel.background = element_blank()
        ) +
  ylab("run time") +
  xlab("movies")
```



As we can see, there is only a small fraction of movies that are close to the mean. If we now consider the distance of each dot from the red line, we know how much each dot, i.e. each movie, deviates from the mean. The standard deviation tells us how much the runtime deviates on average. To be more specific, to compute the standard deviation by hand you would:

- subtract the mean runtime from the runtime of each movie and square it, i.e. $deviations = (runtime\_min - mean_{all\ movies})^2$,
- then we take the sum of all deviations and divide it by the number of movies minus 1, i.e. $\frac{\sum(deviations)}{250-1}$, and because we squared the deviations,
- we take the square root of this score, i.e. $\sqrt{\frac{\sum(deviations)}{250-1}}$.

We could compute this by hand if we wanted, but it is much simpler to use the function`sd()` to achieve the same. The result shows that movies tend to be about 32 minutes longer or shorter than the average movie. This seems quite long.

```
sd(imdb_top_250$runtime_min)
## [1] 32.63701
```

However, we have to be aware that this score is also influenced by the outliers we detected before. As such, if standard deviations in your data appear quite large, it likely is due to outliers and you should investigate further. Needless to say, plotting your data will undoubtedly help to diagnose any outliers.

8.3.3 Dealing with outliers

In this Chapter, I referred to outliers many times but never eluded to the aspects of handling them. Dealing with outliers is similar to dealing with missing data. It is not quite as straightforward as one might think.

In a first step, we need to determine which values count as an outlier. Aguinis et al. (2013) reviewed 232 journal articles and found that scholars had defined outliers in 14 different ways, used 39 different techniques to detect them and applied 20 different strategies to handle them. It would be impossible to work through all different options in this book. However, I want to offer two options that have been frequently considered in publications in the field of Social Sciences:

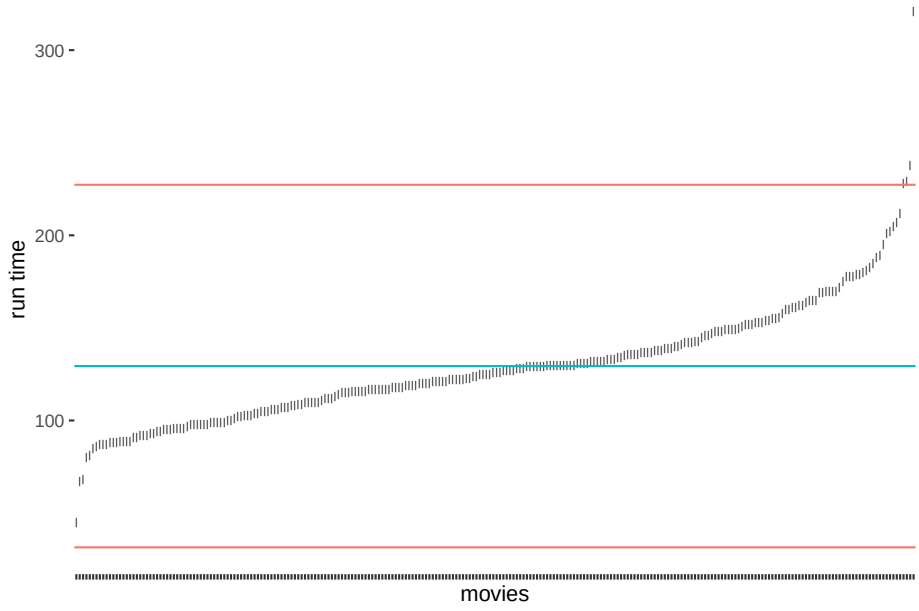
- the standard deviation
- the inter-quartile range

8.3.3.1 Detecting outliers using the standard deviation

A very frequently used approach to detecting outliers is the use of the standard deviation. Usually, scholars use multiples of the standard deviation to determine thresholds. For example, a value that lies 3 standard deviations

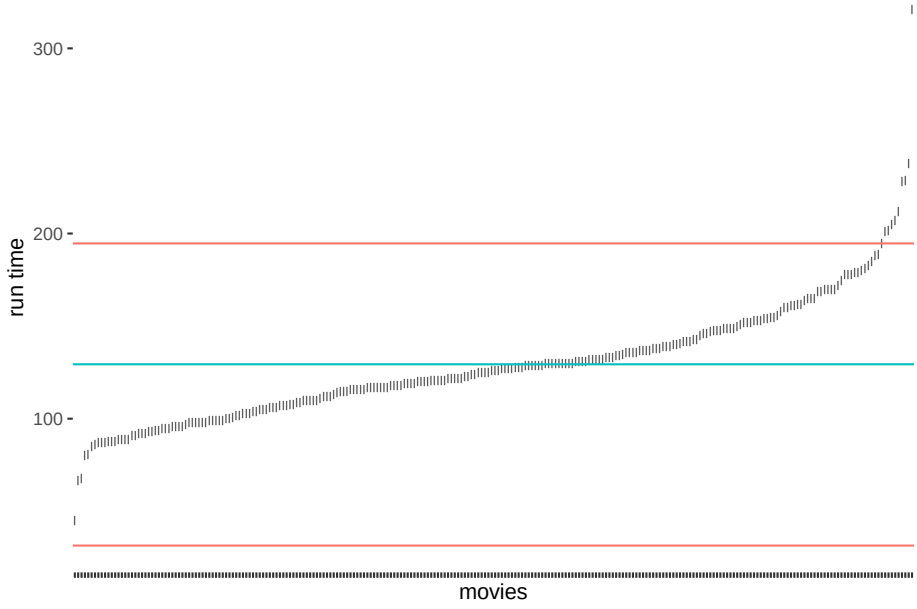
above or below the mean could be categorised as an outlier. Unfortunately, there is quite some variability regarding how many multiples of the standard deviation counts as an outlier. Some authors might use 3, others might settle for 2 (see also Leys et al. (2013)). Let's stick with the definition of 3 standard deviations to begin with. We can revisit our previous plot and add lines which show the thresholds above and below the mean.

```
# Compute the mean and standard deviation
runtime_mean <- mean(imdb_top_250$runtime_min)
sd_upper <- runtime_mean + 3 * sd(imdb_top_250$runtime_min)
sd_lower <- runtime_mean - 3 * sd(imdb_top_250$runtime_min)

# Create our plot
imdb_top_250 %>%
  select(title, runtime_min) %>%
  ggplot(aes(x = reorder(title, runtime_min), y = runtime_min)) +
  geom_point(shape = 124) +
  geom_hline(aes(yintercept = runtime_mean, colour = "red"), show.legend = FALSE) +
  geom_hline(aes(yintercept = sd_upper, color = "blue"), show.legend = FALSE) +
  geom_hline(aes(yintercept = sd_lower, color = "blue"), show.legend = FALSE) +

  theme(axis.text.x = element_blank(),
        panel.grid.major = element_blank(),
        panel.background = element_blank()
        ) +
  ylab("run time") +
  xlab("movies")
```



The results suggests that only very few outliers would be detected if we chose these thresholds. Especially 'Sherlock Jr.', the shortest movie in our dataset would not classify as an outlier. How about we choosen two standard deviations instead?

```
# Compute the mean and standard deviation
runtime_mean <- mean(imdb_top_250$runtime_min)
sd_upper <- runtime_mean + 2 * sd(imdb_top_250$runtime_min)
d_lower <- runtime_mean - 2 * sd(imdb_top_250$runtime_min)

# Create our plot
imdb_top_250 %>%
  select(title, runtime_min) %>%
  ggplot(aes(x = reorder(title, runtime_min), y = runtime_min)) +
  geom_point(shape = 124) +
  geom_hline(aes(yintercept = runtime_mean, colour = "red"), show.legend = FALSE) +
  geom_hline(aes(yintercept = sd_upper, color = "blue"), show.legend = FALSE) +
  geom_hline(aes(yintercept = sd_lower, color = "blue"), show.legend = FALSE) +

  theme(axis.text.x = element_blank(),
        panel.grid.major = element_blank(),
        panel.background = element_blank()
        ) +
  ylab("run time") +
  xlab("movies")
```



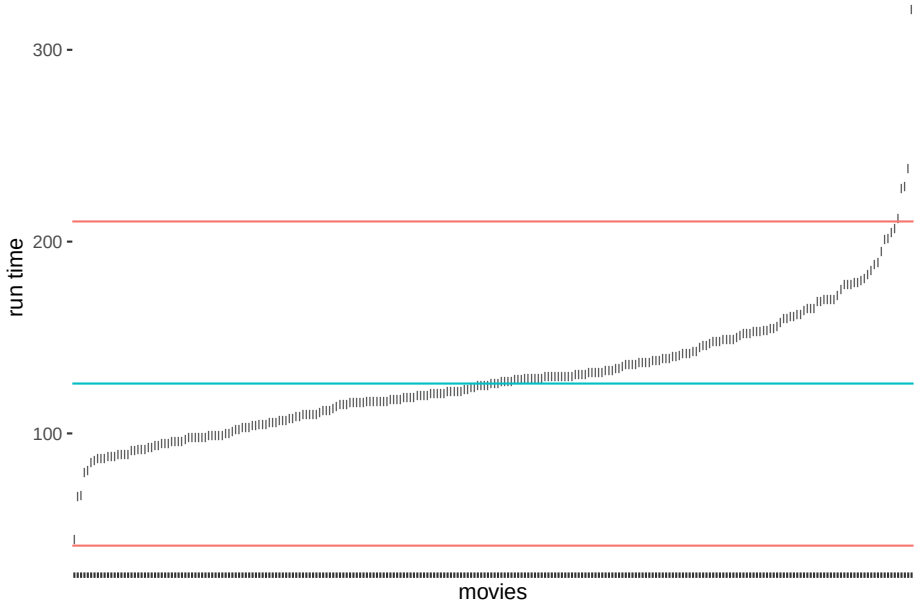
As we would expect, we identify some more movies as being outliers. Still, it feels rather arbitrary to choose a threshold of our liking. Despite its popularity, there are additional problems with this approach:

- outliers affect our mean and standard deviation too,
- since we use the mean, we assume that our data is normally distributed, and
- in smaller samples, this approach might result in not identifying outliers at all (despite their presence) (Leys et al. (2013), p. 764)

Leys et al. (2013) propose an alternative approach, based on the fact that medians are much less vulnerable to outliers than the mean. Similarly to the standard deviation, it is possible to calculate thresholds using the 'median absolute deviation' (MAD). Best of all, the function `mad()` in R does this automatically for us. Leys et al. (2013) suggest to use 2.5 times the MAD as a threshold. However, if we want to compare directly how well this option does against the standard deviation, we should `3` again.

```
# Compute the median and thresholds
runtime_median <- median(imdb_top_250$runtime_min)
mad_upper <- runtime_median + 3 * mad(imdb_top_250$runtime_min)
mad_lower <- runtime_median - 3 * mad(imdb_top_250$runtime_min)
```

```
# Create our plot
imdb_top_250 %>%
  select(title, runtime_min) %>%
  ggplot(aes(x = reorder(title, runtime_min), y = runtime_min)) +
  geom_point(shape = 124) +
  geom_hline(aes(yintercept = runtime_median, colour = "red"), show.legend = FALSE) +
  geom_hline(aes(yintercept = mad_upper, color = "blue"), show.legend = FALSE) +
  geom_hline(aes(yintercept = mad_lower, color = "blue"), show.legend = FALSE) +

  theme(axis.text.x = element_blank(),
        panel.grid.major = element_blank(),
        panel.background = element_blank()
        ) +
  ylab("run time") +
  xlab("movies")
```



Compared to our previous results, we notice that the median approach was much better in detecting outliers at the upper range of `runtim_min`. Because the median is not affected so much by the five hour-long movie, the results have improved. Still, we would not classify the outlier at the bottom for the shortest movie in the data. If we chose the criterion of 2.5*MAD, we would also get this outlier (see Figure 8.4).

Which approach to choose can be informed by the normality of your data. If your data is normally distributed, the mean and median would be very close

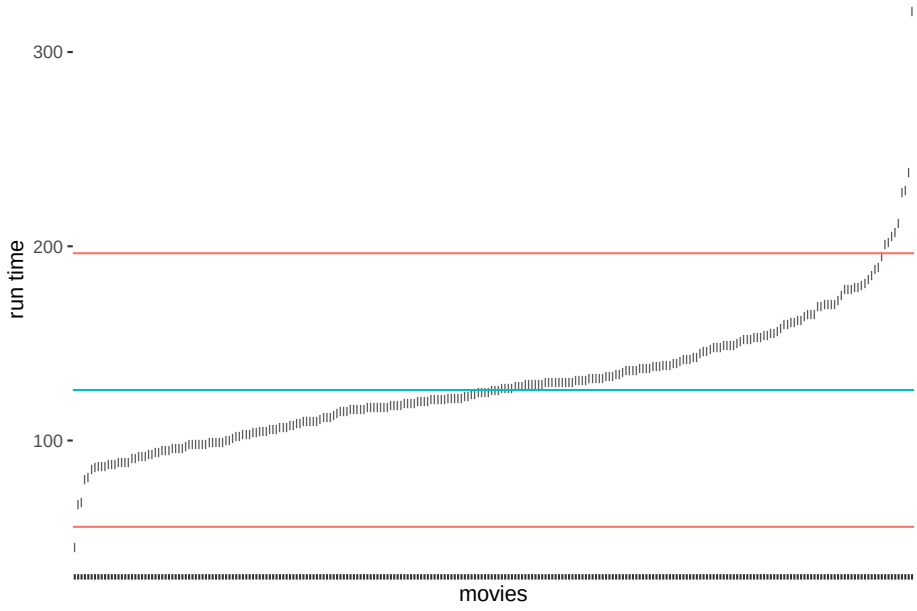


Figure 8.4: Outlier detection via MAD using '2.5 * MAD' as a threshold

to each other and the results from both approach would return very similar results. However, if your data is not normally distributed, it might be better to classify outliers using the median.

8.3.3.2 Detecting outliers using the interquartile range

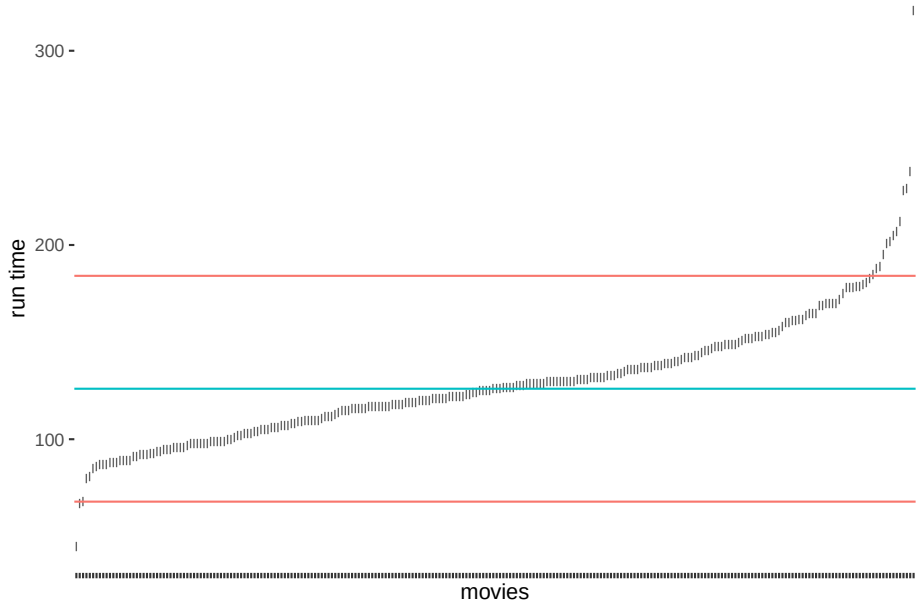
Another approach to classify outliers is the use of the interquartile range (IQR). This one is used in boxplots and creates the dots at its ends to indicate any outliers. This approach is very easy to implement because the computation of the IQR is simple:

$$IQR = Q_3 - Q_1$$

Therefore, we can create new thresholds for the detection of outliers. For IQR it is common to use '1.5 * IQR' as the lower and upper thresholds.

```
# Compute the median and thresholds
runtime_median <- median(imdb_top_250$runtime_min)
iqr_upper <- runtime_median + 1.5 * IQR(imdb_top_250$runtime_min)
iqr_lower <- runtime_median - 1.5 * IQR(imdb_top_250$runtime_min)

# Create our plot
imdb_top_250 %>%
  select(title, runtime_min) %>%
```

```
ggplot(aes(x = reorder(title, runtime_min), y = runtime_min)) +
geom_point(shape = 124) +
geom_hline(aes(yintercept = runtime_median, colour = "red"), show.legend = FALSE) +
geom_hline(aes(yintercept = iqr_upper, color = "blue"), show.legend = FALSE) +
geom_hline(aes(yintercept = iqr_lower, color = "blue"), show.legend = FALSE) +

theme(axis.text.x = element_blank(),
      panel.grid.major = element_blank(),
      panel.background = element_blank()
      ) +
ylab("run time") +
xlab("movies")
```



As we can tell, the IQR detects much more outliers for our data than any of the previous methods. The outliers we find here are the same as shown in Figure 8.1.

For our data it seems that the MAD by Leys et al. (2013) produced the 'best' selection. However, we have to acknowledge that these classifications will always be subjective, because the decision of how we position the thresholds is still depending on the researcher's choice.

8.3.3.3 Removing or replacing outliers

Now that we have identified our outliers we are confronted with the question what we should do with them. Similar to missing data (see Chapter 7.6) we can either remove them or replace them with other values. While removal is a fairly simple task, replacing it with other 'reasonable' values implies that we need to find techniques to create such values. As you may remember, we were confronted with a similar problem before when we looked into missing data (Chapter 7.6. The same techniques, especially multiple imputation (see Cousineau and Chartier, 2010), can be used for such scenarios as well.

Irrespective of whether we remove or replace outliers, we somehow need to single them out of the crowd. Since the MAD strategy worked well for our data, we can use the thresholds we defined before, i.e. `mad_upper` and `mad_lower`. Therefore, an observation (i.e. a movie) is considered as an outlier if:

- its value lies above `mad_upper`, or
- its value lies below `mad_lower`

It becomes clear that we somehow need to define a condition, because if it is an outlier, it should be labelled as one, if not then it should not be labelled as one. Ideally we want a new column in our dataset which indicates whether a movie is an outlier (i.e. outlier = TRUE) or not (outlier = FALSE). R offers a way for us to express such conditions with the function `ifelse()`. It has the following structure:

```
ifelse(condition, TRUE, FALSE)
```

Let's formulate a sentence that describes our scenario as an `ifelse()` function:

- If a movie's `runtime_min` is longer than `mad_upper`, or
- if a movie's `runtime_min` is lower than `mad_lower`,
- classify this movie as an outlier (i.e. `TRUE`),
- otherwise classify this movie as not being an outlier (i.e. `FALSE`).

We already know from Chapter 5.1 how to use logical and arithmetic operators. All we have to do is put them together in one function call.

```
imdb_top_250 <- imdb_top_250 %>%
  mutate(outlier = ifelse(runtime_min > mad_upper | runtime_min < mad_lower,
                          TRUE, FALSE))
```

Since we have a classification we can more thoroughly inspect our outliers and see which movies are the ones that are lying outside our defined norm. We can `arrange()` them by `runtime_min`.

```
imdb_top_250 %>%
  filter(outlier == "TRUE") %>%
  select(title, runtime_min, outlier) %>%
  arrange(runtime_min)
## # A tibble: 10 x 3
##    title                                          runtime_min outlier
##    <chr>                                                <dbl> <lgl>
##  1 Sherlock Jr.                                            45 TRUE
##  2 The Lord of the Rings: The Return of the King          201 TRUE
##  3 The Godfather: Part II                                 202 TRUE
##  4 Andrei Rublev                                          205 TRUE
##  5 Seven Samurai                                          207 TRUE
##  6 Ben-Hur                                                212 TRUE
##  7 Lawrence of Arabia                                     228 TRUE
##  8 Once Upon a Time in America                            229 TRUE
##  9 Gone with the Wind                                     238 TRUE
## 10 Gangs of Wasseypur                                     321 TRUE
```

The list of movies contains some of the most iconic Hollywood films ever shown on screen. I think we can agree that most of them are truly outside the norm of regular movies, not just in terms of runtime.

From here it is simple to remove these movies (i.e. keep the movies that are not outliers) or set their values to `NA` by writing the following lines of code. We replace the values with `NA` we can the continue with one of the techniques demonstrated for missing values (Chapter 7.6.

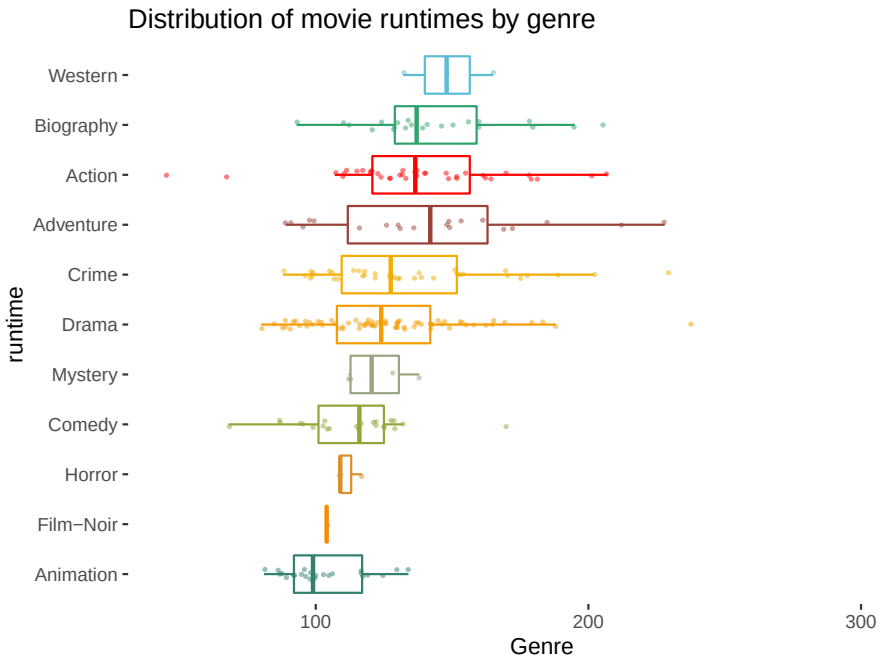
```
# Keep all movies that are not outliers
imdb_top_250 %>%
  filter(outlier == "FALSE")

# Replace values with NA
imdb_top_250 %>%
  mutate(runtime_min = replace(runtime_min, outlier == "TRUE", NA))
```

The `replace()` function is very intuitive to use. It first needs to know where you want to replace a value (`runtime_min`), then what the condition for replacing is (`outlier == "TRUE"`), and lastly, which value should be put instead of the original one (`NA`).

As you hopefully noticed, understanding your data requires some effort, but it is important to know your data well before proceeding to any further analysis.

You can experiment with different data visualisations and design them in a way that best reflect the message you want to get across. For example, because we have now a separate variable which classifies outliers we can do more with our data visualisation than before and dress it up a bit more nicely.

```
colour_pal <- wesanderson::wes_palette("Darjeeling1", 11, type = "continuous")

imdb_top_250 %>%
  ggplot(aes(x = reorder(genre_01, runtime_min),
             y = runtime_min,
             colour = genre_01)
         ) +
  geom_boxplot(alpha = 0,
               show.legend = FALSE) +
  geom_jitter(width = 0.1,
              size = 0.5,
              alpha = 0.5,
              show.legend = FALSE) +
  scale_color_manual(values = colour_pal) +
  coord_flip() +
  theme(panel.background = element_blank())+
  xlab("runtime") +
  ylab("Genre") +
  ggtitle("Distribution of movie runtimes by genre")
```


Distribution of movie runtimes by genre
Western
Biography
Action
Adventure
Crime
Drama
Mystery
Comedy
Horror
Film−Noir
Animation
runtime
100
200
300
Genre

Chapter 9

Correlations

Sometimes counting and measuring means, medians and standard deviations is not enough, because they are all based on a single variable. Instead, we might have questions related to the relationship of two or more variables. In this section we will explore how correlations can (and kind of cannot - see Chapter 9.4.3) provide insights into the following questions:

- Do movie viewers agree with critics regarding the rating of movies?
- Do popular movies receive more votes from users than less popular movies?
- Do movies with more votes also make more money?

There are many different ways of how one can compute the correlation and it partially depends on the type of data you want to relate. The 'Pearson's correlation' is by far the most frequently used correlation technique for data that is normally distributed. On the other hand, if data is not normally distributed, we can opt for the 'Spearman's rank' correlation. One could argue that the relationship between these two correlations is like the mean (Pearson) to the median (Spearman). Both approaches require numeric values to be computed properly. If our data is ordinal, or worse dichotomous (like a logical variables), we have to opt for different options.

Table 9.1: Different ways of computing correlations

Correlation	Used
Pearson	Requires variables to be parametric and therefore numeric
Spearman	Used when data is non-parametric and requires numeric variables
Polychoric	Used when investigating two ordinal variables

Correlation	Used
Tetrachoric	Used when both variables are dichotomous, e.g. 'yes/no', 'True/False'.
Rank-biserial	Used when one variable is dichotomous and the other variable is ordinal

There are many more variations of correlations, which you can explore on the website of the package we will use in this chapter: `correlation`. We will primarily focus on Pearson, Spearman and the Chi-Squared test, because they are the most commonly used correlation types in academic publications to understand the relationship between two variables. In addition, we also look at 'partial correlations', which allow us to introduce a third variable into this mix.

9.1 Parametric or non-parametric: That is the question

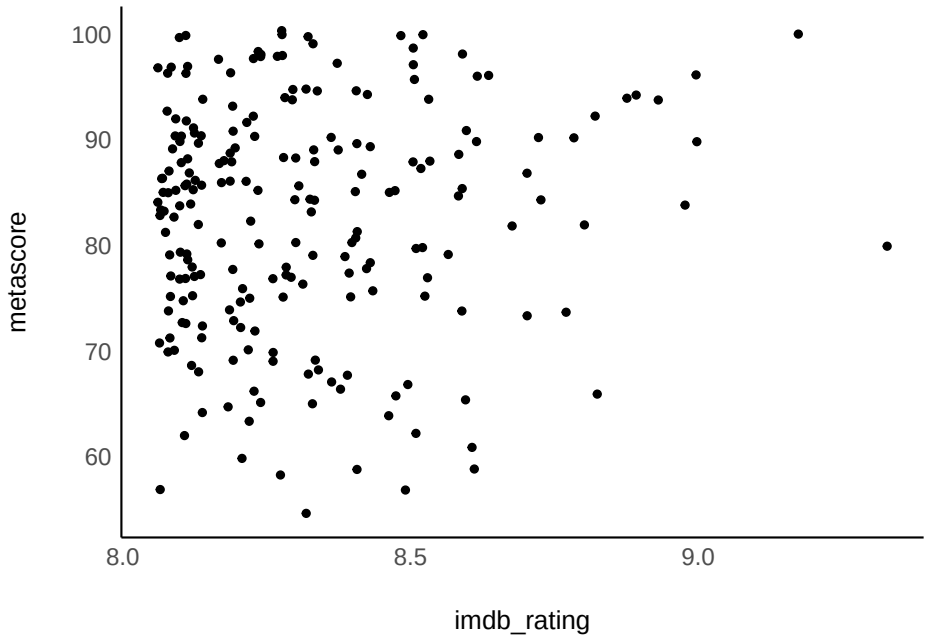
`TODO: ADD SECTION`

9.2 Plotting correlations

Since correlations only show the relationship between two variables, we can easily put one variable onto the x axis and one variable onto the y axis creating a so-called 'scatterplot'. We used the functions to create scatterplots before, i.e. `geom_point()` and `geom_jitter`. Let's try to answer our first research question, i.e. whether regular movie viewers and critics (people who review movies as a profession) rate the top 250 in the same way. One assumption could be that it does not matter whether you are a regular movie viewer or someone who does it professionally. After all, we are just human beings. A counter thesis could be that critics have a different perspective on movies and might use different evaluation criteria. Either way, we first need to identify the two variables of interest:

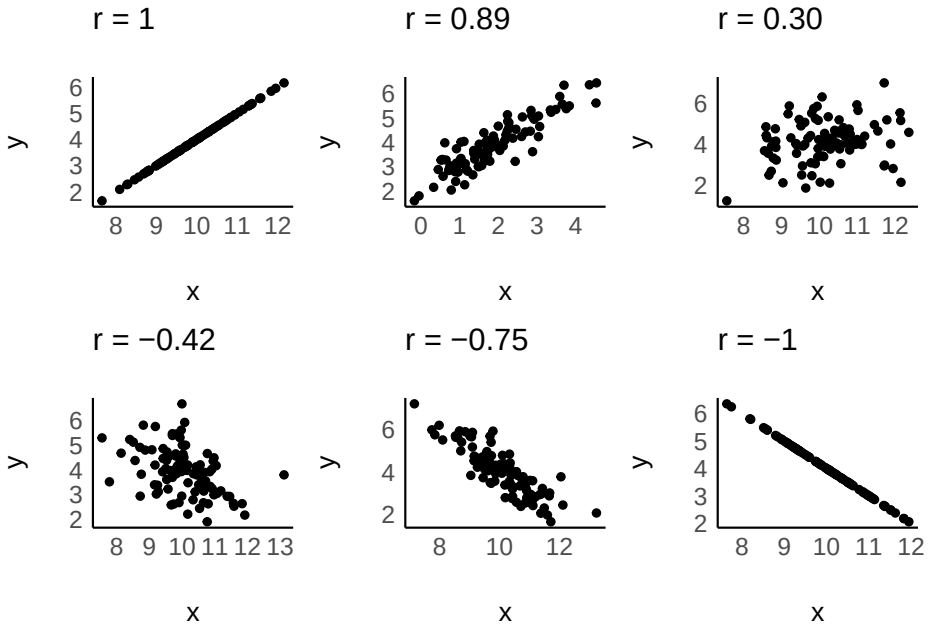
- `imdb_rating` is based on IMDb users
- `metascore` is based on movie critics

```
imdb_top_250 %>%
  filter(!is.na(metascore)) %>%
  ggplot(aes(imdb_rating, metascore)) +
  geom_jitter() +
  see::theme_modern()
```



The results from our scatterplot are, well, somewhat random. We can see that some movies receive high `imdb_ratings` as well as high `metascores`. However, there are also some movies that receive high `imdb_ratings`, but low `metascores`. Overall, the points look like they are randomly scattered all over our canvas. The only visible pattern we can notice is that there are more movies at the lower end of the rating system relative to all the movies in the top 250. In fact, ther are only 2 movies which actually received an IMDb rating of over 9. Be aware, `geom_jitter()` makes it look like there were more than 9!).

Since correlations only explain linear relationships, a perfect correlation would be represented by a straight line. Consider the following examples of correlations:



A correlation can be either positive or negative and its score (i.e. `r` in case of Pearson) can range from -1 to 1:

- -1 defines a perfectly negative correlation,
- 0 defines no correlation (completely random), and
- 1 defines a perfectly positive correlation.

In other words, the further the score is away from zero, the stronger is the relationship between variables. We also have benchmarks which we can use to asses the strength of a relationship, for example the one by Cohen (1988). The strength of the relationship is also called 'effect size'. Table 9.2 shows the relevant benchmarks. Note that effect sizes are always provided as absolute figures. Therefore, -0.4 would also count as a moderate relationship.

Table 9.2: Assessing effect size of relationships according to Cohen (1988)

effect size	interpretation
$r < 0.1$	very small
$0.1 \leq r < 0.3$	small
$0.3 \leq r < 0.5$	moderate
$r \geq 0.5$	large

If we compare the plot from our data with the sample plots, we would come to the conclusion that the relationship is weak, and therefore the `r` must be close to zero as well. We can test this with the Pearson correlation.

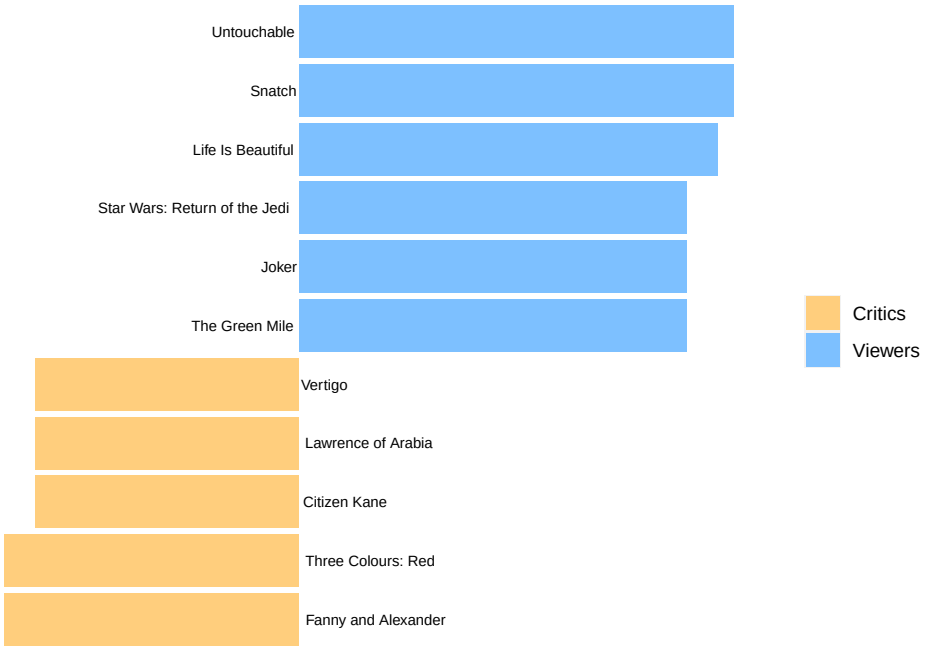
```
library(correlation)

imdb_top_250 %>%
  select(imdb_rating, metascore) %>%
  correlation()
## # Correlation Matrix (pearson-method)
##
## Parameter1   | Parameter2 |    r |        95% CI | t(214) |     p
## ------------------------------------------------------------------
## imdb_rating |  metascore | 0.08 | [-0.06, 0.21] |   1.11 | 0.270
##
## p-value adjustment method: Holm (1979)
## Observations: 216
```

Indeed, our analysis reveals that the effect size is very small ($r = 0.08 < 0.1$). Therefore, critics appear to rate movies differently than regular movie viewers. This triggers an interesting follow-up question: Which movie is the most controversial, i.e. where is the difference between `imdb_rating` and `metascore` the highest?

We can answer this question with the tools we already know. We create a new variable to subtract the `metascore` from the `imdb_rating` and plot it. We have to make sure both scales are the same length. The variable `imdb_rating` ranges from 0-10, but the `metascore` ranges from 0-100. As such, I used the percentile score instead by dividing the scores by 10 and 100 respectively. Since plotting 250 movies would have been too much and would also not help us find the answer to our question, I chose arbitrary values to pick only those movies with the highest differences in scores. Feel free to adjust the filter to your liking to see more or less movies.

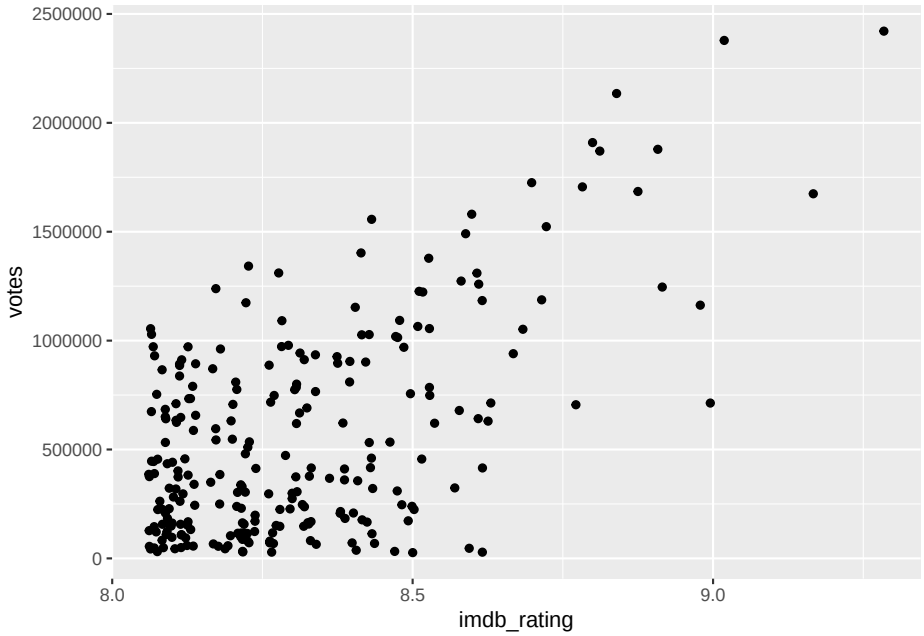
```
plot <- imdb_top_250 %>%
  mutate(r_diff = imdb_rating/10 - metascore/100) %>%
  filter(!is.na(r_diff) & r_diff >= 0.25 | r_diff <= -0.165)

plot %>%
  ggplot(aes(x = reorder(title, r_diff), y = r_diff, label = title)) +
  geom_col(aes(fill = ifelse(r_diff > 0, "Viewers", "Critics"))) +

  geom_text(aes(x = title, y = 0,
            label = title),
            size = 2.5,
```

```
            # to adjust the labels of the plot
            vjust = ifelse(plot$r_diff >= 0, 0.5, 0.5),
            hjust = ifelse(plot$r_diff >= 0, 1.05, -0.05)
            ) +
  coord_flip() +

  # Cleaning up the plot to make it look more readable and colourful
  scale_fill_manual(values = c("#FFCE7D", "#7DC0FF")) +
  theme(axis.title = element_blank(),
        plot.background = element_blank(),
        panel.background = element_blank(),
        axis.text = element_blank(),
        axis.ticks = element_blank(),
        legend.title = element_blank()
        )
```



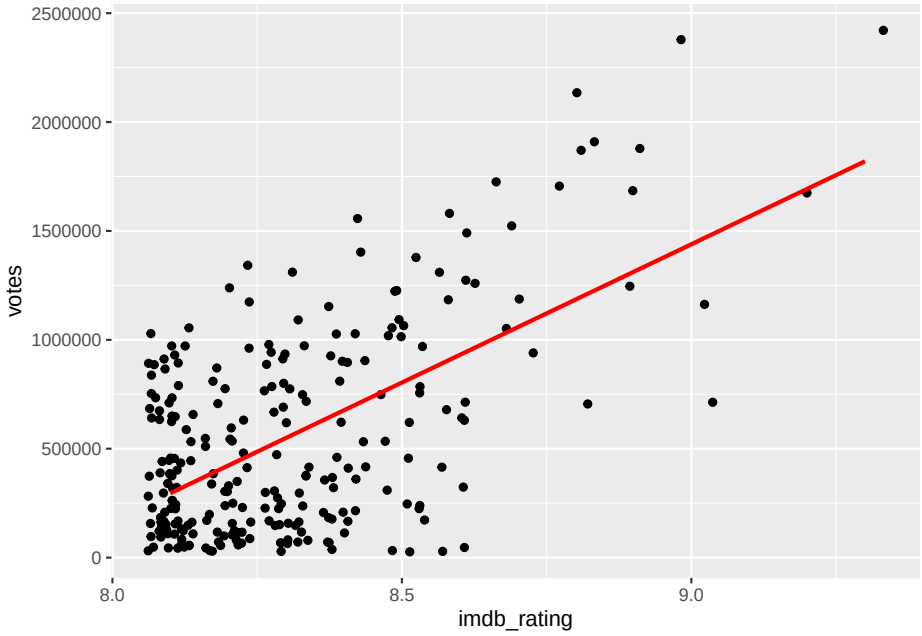
Another question we posed at the beginning was: Do popular movies receive more votes from users than less popular movies? Our intuition might say 'yes'. More popular movies are likely seen by more people, which makes them more popular. Consequently, the more people have seen a movie, the more likely they might vote for this movie. For less popular movies, the oppposite should be true. Let's create another scatterplot.

```
imdb_top_250 %>%
  ggplot(aes(x = imdb_rating, y = votes)) +
  geom_jitter()
```



The scatterplot shows some positive trend. Often, it can be touch to see the trend clearly. In order to impove our plot we can make use of the function `geom_smooth()`, which can help us draw a straight line that fits our data points the best. We need to set the `method` for drawing the line to `lm`, which stands for linear model. Remember, correlations assume a linear relationship between two variables.

```
imdb_top_250 %>%
  ggplot(aes(x = imdb_rating, y = votes)) +
  geom_jitter() +
  geom_smooth(method = "lm",
              formula = y ~ x,
              se = FALSE,
              colour = "red")
```



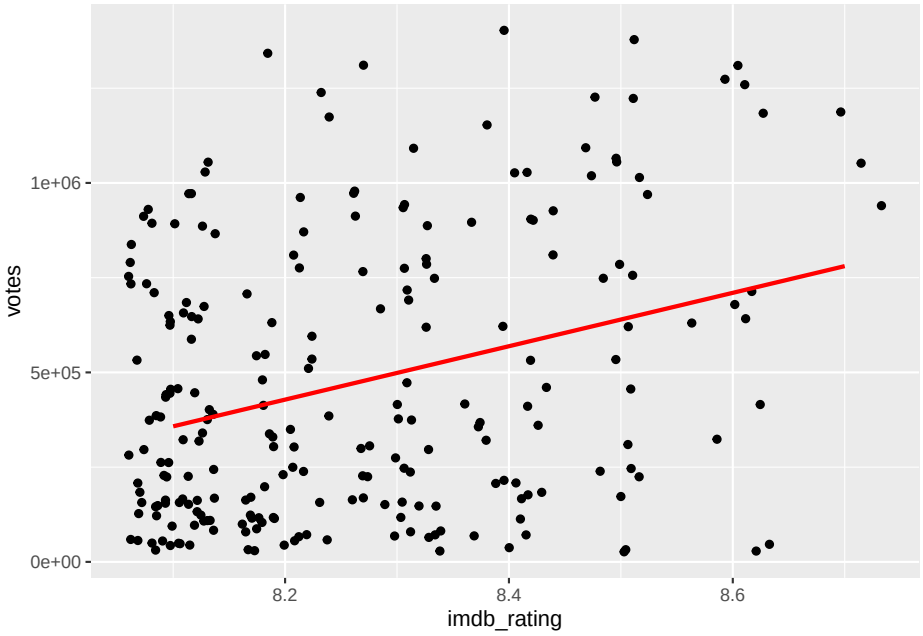
Another problem we face with this plot (and correlations) are extreme values, i.e. outliers. We already know that outliers tend to cause trouble for our analysis (see Chapter 8.3.3 and in correlations they can affect the strength of relationships. If we compute the the Pearson correlation with and without some of the outliers, the differences are very significant. Be aware, since we work with two variables at the same time, we should consider outliers on both. There our `filter()` will have to include two conditions.

```
# The correlation with outliers
imdb_top_250 %>%
  select(imdb_rating, votes) %>%
  correlation()
## # Correlation Matrix (pearson-method)
##
## Parameter1  | Parameter2 |    r |       95% CI | t(248) |         p
## ------------------------------------------------------------------
## imdb_rating |      votes | 0.59 | [0.51, 0.67] |  11.57 | < .001***
##
## p-value adjustment method: Holm (1979)
## Observations: 250

# Define outliers
votes_out <- 1.5*IQR(imdb_top_250$votes) + median(imdb_top_250$votes)
imdb_r_out <- 1.5*IQR(imdb_top_250$imdb_rating) + median(imdb_top_250$imdb_rating)
```

```
# The correlation with outliers (based on 1.5 * IQR)
imdb_top_250 %>%
  filter(votes <  votes_out & imdb_rating < imdb_r_out) %>%
  select(imdb_rating, votes) %>%
  correlation()
## # Correlation Matrix (pearson-method)
##
## Parameter1  | Parameter2 |    r |       95% CI | t(230) |         p
## ------------------------------------------------------------------
## imdb_rating |      votes | 0.31 | [0.19, 0.42] |   4.94 | < .001***
##
## p-value adjustment method: Holm (1979)
## Observations: 232
```

While both correlations are highly significant ($p < 0.01$), the drop in `r` from 0.59 to 0.31 is substantial. When we plot the data again, we can see that the dots are fairly randomly distributed across the plotting area. Thanks to `geom_smooth` we get an idea of a slight positive relationship between these two variables.

```
imdb_top_250 %>%
  filter(votes <  votes_out & imdb_rating < imdb_r_out) %>%
  ggplot(aes(x = imdb_rating, y = votes)) +
  geom_jitter() +
  geom_smooth(method = "lm",
              formula = y ~ x,
              se = FALSE,
              colour = "red")
```



In conclusion, while there is some relationship between the rating and the number of votes, it is by far not as strong as we might have thought, especially after removing outliers, which had a considerable effect on the effect size.

9.3 Significance: A way to help you judge your findings

One of the most common pitfalls of novice statisticians is related to the interpretation of what counts as significant and not significant. Most basic tests offer a 'p value', which stands for 'probability value'. The `p` value can range from 1 (for 100%) to 0 (for 0%) and implies:

- p = 1, there is a 100% chance that the result is a pure coincidence
- p = 0, there is a 0% chance that the results is a pure coincidence, i.e. we can be certain this is not just luck.

Technically, we would not find that `p` is ever truly zero, and instead denote very small `p` values with `p < 0.01` or even `p< 0.001`.

There are also commonly considered thresholds for the `p` value:

- $p > 0.05$, the result is not significant. There is chance of 5% that our finding is a pure coincidence.

- p ≤ 0.05 , the result is significant.
- p ≤ 0.01, the result is highly significant.

We will cover more about the `p` value in Chapter 10 and Chapter 11. For now, it is important to know that a significant correlation is one that we should look at more closely. Usually, correlations that nor significant suffer from low effect sizes. However, different samples can lead to different effect sizes and different significant levels. Consider the following examples:

```
## # Correlation Matrix (pearson-method)
##
## Parameter1 | Parameter2 |    r |        95% CI | t(2) |     p
## ------------------------------------------------------------
## x          |          y | 0.77 | [-0.73, 0.99] | 1.73 | 0.225
##
## p-value adjustment method: Holm (1979)
## Observations: 4
## # Correlation Matrix (pearson-method)
##
## Parameter1 | Parameter2 |    r |       95% CI | t(10) |       p
## ------------------------------------------------------------------
## x          |          y | 0.77 | [0.36, 0.93] |  3.87 | 0.003**
##
## p-value adjustment method: Holm (1979)
## Observations: 12
```

In both examples $r = 0.77$, but the sample sizes are different (4 vs 12) and the `p` values differ as well. In the first example $p = 0.225$, which means the relationship is not significant, while in the second example, we find that $p < 0.01$ and is therefore highly significant. As a general rule, we find that the bigger the sample, the more likely we find significant results, even though the effect size is small. Therefore, it is important to interpret correlations base on at least three factors:

- the `p` value, i.e. significance level
- the `r` value, i.e. the effect size, and
- the sample size.

The interplay of all three can help determine whether a relationship is truly important. Therefore, when we include correlation tables in publications, we have to make sure we provide information about all three.

It is common that we do not only compute correlations for two variables at a time, instead, we can do this for multiple variables simultaneously.

```
imdb_top_250 %>%
  select(imdb_rating, metascore, year, votes, gross_in_m) %>%
  correlation()
## # Correlation Matrix (pearson-method)
##
## Parameter1  | Parameter2 |     r |         95% CI |     t |  df |         p
## ------------------------------------------------------------------------------
## imdb_rating |  metascore |  0.08 | [-0.06,  0.21] |  1.11 | 214 | 0.539
## imdb_rating |       year |  0.03 | [-0.10,  0.15] |  0.42 | 248 | 0.678
## imdb_rating |      votes |  0.59 | [ 0.51,  0.67] | 11.57 | 248 | < .001***
## imdb_rating | gross_in_m |  0.21 | [ 0.07,  0.33] |  3.07 | 213 | 0.010**
## metascore   |       year | -0.41 | [-0.52, -0.30] | -6.63 | 214 | < .001***
## metascore   |      votes | -0.25 | [-0.37, -0.12] | -3.76 | 214 | 0.001**
## metascore   | gross_in_m | -0.13 | [-0.27,  0.01] | -1.83 | 193 | 0.207
## year        |      votes |  0.37 | [ 0.26,  0.47] |  6.29 | 248 | < .001***
## year        | gross_in_m |  0.36 | [ 0.23,  0.47] |  5.58 | 213 | < .001***
## votes       | gross_in_m |  0.56 | [ 0.46,  0.64] |  9.79 | 213 | < .001***
##
## p-value adjustment method: Holm (1979)
## Observations: 195-250
```

If you have seen correlation tables before, you might find that `correlation()` does not produce the classic table by default. If you want it to look like the tables in publications, which are more compact but offers less information, you can use the function `summary()`.

```
imdb_top_250 %>%
  select(imdb_rating, metascore, year, votes, gross_in_m) %>%
  correlation() %>%
  summary()
## # Correlation Matrix (pearson-method)
##
## Parameter   | gross_in_m |   votes |     year | metascore
## ----------------------------------------------------------
## imdb_rating |     0.21** | 0.59*** |     0.03 |      0.08
## metascore   |      -0.13 | -0.25** | -0.41*** |
## year        |    0.36*** | 0.37*** |          |
## votes       |    0.56*** |         |          |
##
## p-value adjustment method: Holm (1979)
```

This table also provides an answer to our final question, i.e. do movies with more votes earn more money. It appears as if this is true, because $r = 0.56$ and $p < 0.001$. In the classic correlation table you also see `*`. These stand for the difference significant levels:

- *, i.e. $p < 0.05$
- **, i.e. $p < 0.01$
- ***, i.e. $p < 0.001$ (although you might find some do not use this as a separate level)

In short, the more * there are attached to each value, the more significant a result. In other words, relationships with many * likely repeat if we collect data again.

9.4 Limitations of correlations

Correlations are useful, but only to some extend. The three most common limitations you should be aware of are:

- Correlations are not causal relationships
- Correlations can be spurious
- Correlations might only appear in sub-samples of your data

9.4.1 Correlations are not causal relationships

Correlations do not offer insights into causality, i.e. whether change in one variable causes change in the other variable. Correlations only provide insights into whether these two variables tend to change when one of them changes. Still, sometimes we can infer such causality by the nature of the variables. For example, in countries with heavy rain, more umbrellas are sold. It is apparent that buying more umbrellas will not cause more rain, but if there is more rain in a country, we rightly assume that there is a higher demand for umbrellas. If we can theorise the relationship between variables we would rather opt for a regression model instead of a correlation (see Chapter 11.

9.4.2 Correlations can be spurious

Just because we find a relationship between two variables does not necessarily mean that are truly related to each other. Instead, it might be possible that a third variable is the reason for the relationship. We call relationships between variables that are cause by a third variable 'spurious correlations'. This third variable can either be part of our dataset or even something we have not measured at all. The latter case would make it impossible to investigate the relationship further. However, we can always test whether some of our

variables affect the relationship between the two variables of interest. This can be done by using partial correlations. A partial correlation returns the relationship between two variables minus the relationship two a third variable. Figure 9.4.2 depicts this visually. While `a` and `b` appear to be correlated with each other, the correlation might only exist because of their correlation with `x`.

Illustration of a spurious correlation

Let's consider a practical example. We found that `votes` and `gross_in_m` are positively correlated with each other. However, could it be possible that these are affected by the year in which the movies was published. We could assume that movies that appeared later received more votes, because it has become more of a cultural phenomenon to vote about almost everything online[1].

```
# Correlation between variables without considering partial correlations
imdb_top_250 %>%
  select(votes, gross_in_m, year) %>%
  correlation()
## # Correlation Matrix (pearson-method)
##
## Parameter1 | Parameter2 |    r |       95% CI |    t |  df |         p
## ------------------------------------------------------------------------
## votes      | gross_in_m | 0.56 | [0.46, 0.64] | 9.79 | 213 | < .001***
## votes      |       year | 0.37 | [0.26, 0.47] | 6.29 | 248 | < .001***
## gross_in_m |       year | 0.36 | [0.23, 0.47] | 5.58 | 213 | < .001***
##
## p-value adjustment method: Holm (1979)
## Observations: 215-250

# Correlation between variables with consideration of partial correlations
imdb_top_250 %>%
  select(votes, gross_in_m, year) %>%
  correlation(partial = TRUE)
## # Correlation Matrix (pearson-method)
##
## Parameter1 | Parameter2 |    r |       95% CI | t(213) |         p
## ------------------------------------------------------------------
## votes      | gross_in_m | 0.48 | [0.37, 0.58] |   7.98 | < .001***
## votes      |       year | 0.28 | [0.16, 0.40] |   4.31 | < .001***
## gross_in_m |       year | 0.16 | [0.03, 0.29] |   2.43 | 0.016*
##
## p-value adjustment method: Holm (1979)
## Observations: 215
```

The first table reveals that there is a strong relationship between the variables of interest, i.e. between `votes` and `gross_in_m` with an effect size of $r = 0.56$.

[1] If you want to know what people vote on these days, have a look at www.ranker.com

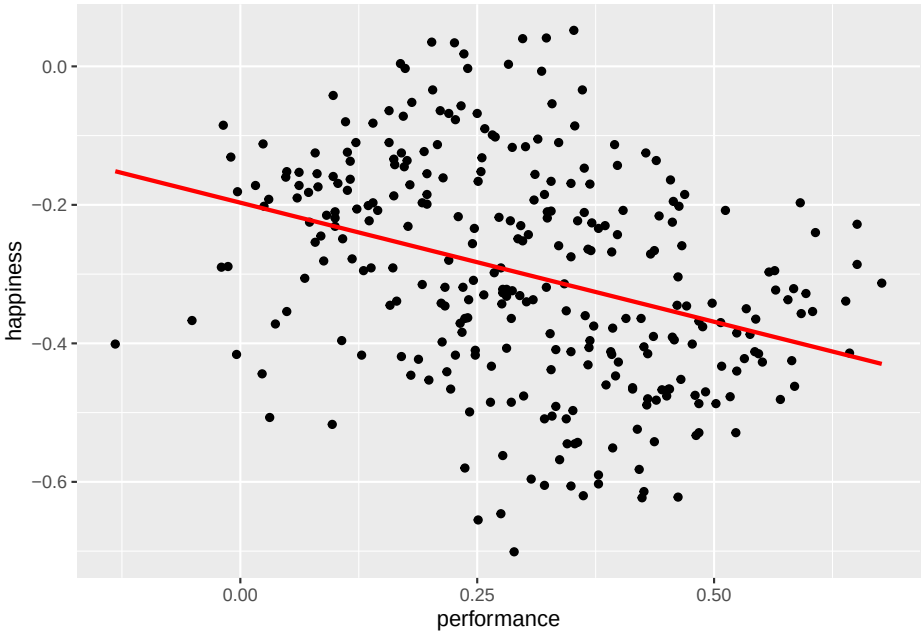
There is also a significant relationship between `votes` and `year`, $r = 0.37$ and between `gross_in_m` and `year`, $r = 0.36$. Thus, we should control for `year` to see how it might affect the relationship between `votes` and `gross_in_m`. The second table, suggests that `year` does not much affect the relationship between `votes` and `gross_in_m`. However, we do notice that the relationship between `gross_in_m` and `year` substantially goes down to $r = 0.16$. It appears, as if the `year` has not managed to impact the relationship between `votes` and `gross_in_m` all that much. Therefore, we can be more confident that this relationship is likely not spurious. However, we can never be fully sure, because we might not have all data that could explain this correlation.

9.4.3 Simpson's Paradox: When correlations betray you

The final limitation is so important that it even has its own name: the 'Simpson's Paradox'. Let's find out what is so paradox about some correlations. For this demonstration we have to make use of a different dataset: `simpson` of the `r4np` package. It contains information about changes in student `performance` and changes in `happiness`. The dataset includes responses from three different groups: `Teachers`, `Students` and `Parents`.

We would assume that an increased in students' `performance` will likely increase the `happiness` of participants. After all, all three have stakes in students's `performance`.

```
simpson %>%
  ggplot(aes(performance, happiness)) +
  geom_point() +
  geom_smooth(method = "lm",
              formula = y ~ x,
              se = FALSE,
              color = "red")
```

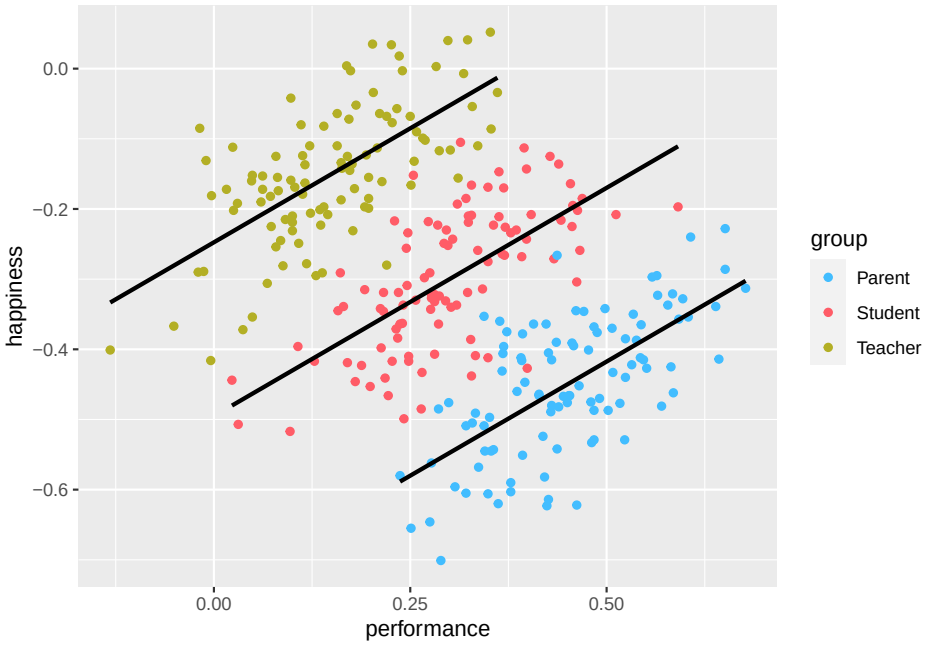



```
simpson %>%
  select(performance, happiness) %>%
  correlation()
## # Correlation Matrix (pearson-method)
##
## Parameter1  | Parameter2 |     r |         95% CI | t(298) |         p
## ------------------------------------------------------------------------
## performance |  happiness | -0.34 | [-0.44, -0.24] |  -6.32 | < .001***
##
## p-value adjustment method: Holm (1979)
## Observations: 300
```

It appears we were terrible wrong. It seems as if `performance` and `happiness` are moderately negatively correlated with each other. Thus, the more a student improves their performance, the less happy teachers, parents and even students are. I hope you agree that this is quite counter-intuitive. However, what could be the cause for such a finding.

The Simpson's paradox explains the phenomenon were correlation might not exist, or even is reversed when considering an entire sample. However, when inspecting sub-samples, different correlations are found. So, what happens if we look at the correlations for each group separately?

```
simpson %>%
  ggplot(aes(performance, happiness, group = group, colour = group)) +
  geom_point() +
  geom_smooth(method = "lm",
              formula = y ~ x,
              se = FALSE,
              color = "black") +

  # Changing colours to make the reference line more visible
  scale_color_manual(values = c("#42BDFF","#FF5C67", "#B3AF25"))
```




```
simpson %>%
  group_by(group) %>%
  select(performance, happiness) %>%
  correlation()
## Adding missing grouping variables: `group`
## # Correlation Matrix (pearson-method)
##
## Group   |  Parameter1 | Parameter2 |    r |       95% CI | t(98) |         p
## ----------------------------------------------------------------------------
## Parent  | performance |  happiness | 0.65 | [0.52, 0.75] |  8.47 | < .001***
## Student | performance |  happiness | 0.65 | [0.52, 0.75] |  8.47 | < .001***
## Teacher | performance |  happiness | 0.65 | [0.52, 0.75] |  8.47 | < .001***
```

```
##
## p-value adjustment method: Holm (1979)
## Observations: 100
```

The results have magically been inverted. Instead of a negative correlation we now find a strong positive correlation for among all groups. This seems to make much sense.

When compute correlations we need to be aware that subsets of our data might show different directions of correlations, or where we found now correlation, there might suddenly be one.

If your study relies on correlations only to detect relationships between variables, which it hopefully does not, it is important to investigate whether the detected or undetected correlations exist. Of course, such an investigation can only be based on data you obtain. The rest remains pure speculation. Nevertheless, as descriptive statistics correlations are very helpful to review the bilateral relationship of your variables. It is often used as a pre-test for regressions (see Chapter 11) and similarly more advanced computations. As a technique to make inferences, the correlation is a good starting point, but should be complemented by other steps, if possible.

Chapter 10

Comparing groups

```
## Rows: 69578 Columns: 6
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (3): country, gender, relationship_status
## dbl (3): age, freedom_of_choice, satisfaction
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Social Sciences is about the study of human beings and their interactions. As such, we frequently want to compare two or more groups of human beings, organisations, teams, countries, etc., with each other to see whether they are similar or different from each other. Sometimes we also want to track individuals over time and see how they may have changed in some way or other. In short, comparing groups is an essential technique to make inferences and helps us better understand the diversity that surrounds us.

If we want to perform a group comparison we have to consider which technique is most appropriate for the data we have. Some of it might be related to the type of data we have collected, other aspects might be linked to the distribution of the data. More specifically, before we apply any statistical technique we have to consider at least the following:

- missing data (see Chapter 7.6),
- outliers (see Chapter 8.3.3, and
- the assumptions made by analytical techniques about our data

While we covered missing data and outliers in previous chapters, we have yet to discuss assumptions. For group comparisons there are three questions we need to answer:

- Is my data parametric or non-parametric? (see Chapter 9.1)
- How many groups do I wish to compare?
- Are these groups paired or unpaired?

In the following we will look at group comparisons for parametric and non-parametric data in each category and use the `wvs_nona` dataset, i.e. the `wvs` dataframe after we performed imputation (see also Chapter 7.6.3). Since we already covered how to test whether data is parametric or non-parametric we will forgo this step out of pure convenience and to remain succinct. We also ignore any potential outliers or missing data. The case studies at the end of the book provide realistic examples of how to perform groups comparisons with a new set of data (see Chapter @ref()). Thus, parametric and non-parametric tests will be demonstrated with the same dataset and the same variables.

10.1 Comparing two groups

The simplest of comparisons is the one where you only have two groups. In our case, we are interested in whether participants perceive `freedom_of_choice` in the same way across different genders and across different countries.

10.1.1 Two Unpaired groups

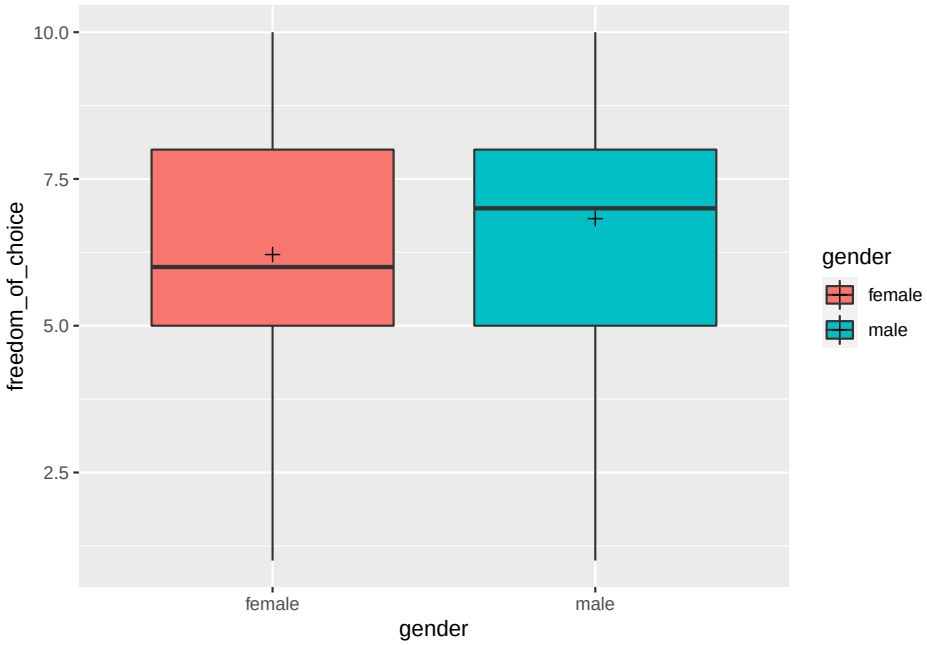
An unpaired group test assumes that the observations in each group are not related to each other, for example that the participants in each group are different individuals.

Our first comparison will be participants from `Egypt` and we want to understand whether male and female citizens in this country perceive that they have `freedom_of_choice`.

We first can compare these two groups using our trusty `geom_boxplot` (or any variation of it).

```
# Only select participants from 'Egypt'
comp <- wvs_nona %>%
  filter(country == "Egypt")

# Compute the mean for each group
group_means <- comp %>%
  group_by(gender) %>%
  summarise(g_mean = mean(freedom_of_choice))

# Create our data visualisation
```

```
comp %>%
  ggplot(aes(x = gender, y = freedom_of_choice, fill = gender)) +
  geom_boxplot() +

  # Add the mean for each group
  geom_point(data = group_means,
             aes(x = gender, y = g_mean),
             shape = 3,
             size = 2
             )
```



While the distribution looks similar, we can notice that the median and the mean (marked by the cross inside the boxplot) are slighter higher for `male` participants. Thus, we could can suspect slight differences between these two groups, but we do not know whether these differences are significant or not. To consider the significance (remember Chapter 9.3) and the effect size (see Table 9.2) we have to perform statistical tests.

Table 10.1 summarises the different tests and functions to perform the group comparison computationally. It is important to note that the parametric test compares the means of two groups, while the non-parametric test compares ranks, especially the median. All tests turn significant if the differences between groups is large enough. Thus, significant results can be read as 'these groups are significantly different from each other'. Of course, if the difference is not significant, the groups are considered to be not different from each other.

Table 10.1: Comparing two unpaired groups (effect size functions from package `effectsize`

Assumption	Test	Function for test	Effect size	Function for effect size
Parametric	T-Test	`t.test()`	Cohen's d	`cohens_d()`
Non-parametric	Mann-Whitney U	`wilcox.test()`	Rank biserial r	`rank_biserial()`

```
# PARAMETRIC COMPARISON
## Perform the comparison
(gcomp <- t.test(freedom_of_choice ~ gender, data = comp))
##
##  Welch Two Sample t-test
##
## data:  freedom_of_choice by gender
## t = -4.7563, df = 1193.2, p-value = 2.212e-06
## alternative hypothesis: true difference in means between group female and group mal
## 95 percent confidence interval:
##  -0.8645067 -0.3595762
## sample estimates:
## mean in group female   mean in group male
##             6.212435             6.824477

## Determine the effect size
(d <- cohens_d(freedom_of_choice ~ gender, data = comp))
## Cohen's d |         95% CI
## --------------------------
## -0.27     | [-0.39, -0.16]
##
## - Estimated using pooled SD.

## Interpret the effect size
interpret_d(d$Cohens_d)
## [1] "small"
## (Rules: cohen1988)

# NON-PARAMETRIC COMPARISON
## Perform the comparison
(gcomp <- wilcox.test(freedom_of_choice ~ gender, data = comp))
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  freedom_of_choice by gender
## W = 149938, p-value = 4.967e-07
## alternative hypothesis: true location shift is not equal to 0
```

```
## Determine the effect size via rank biserial correlation
(d <- rank_biserial(freedom_of_choice ~ gender, data = comp))
## r (rank biserial) |         95% CI
## ---------------------------------
## -0.17             | [-0.23, -0.10]

## Interpret the effect size
interpret_rank_biserial(d$r_rank_biserial, rules = "cohen1988")
## [1] "small"
## (Rules: cohen1988)
```

10.1.2 Two Paired groups

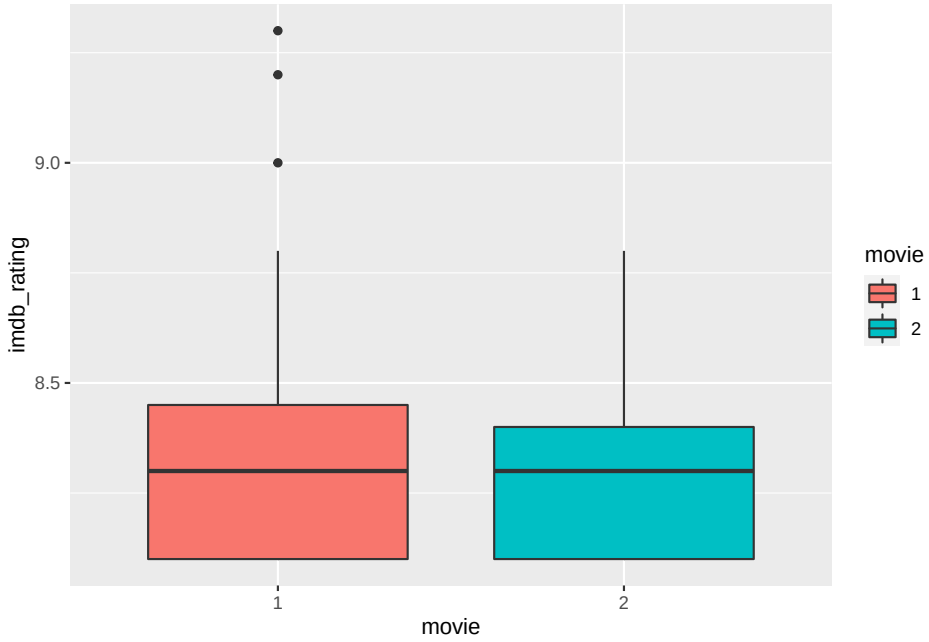
Paired sample test:

- use in longitudinal studies

- used in experimental studies (pre-test vs post-test)

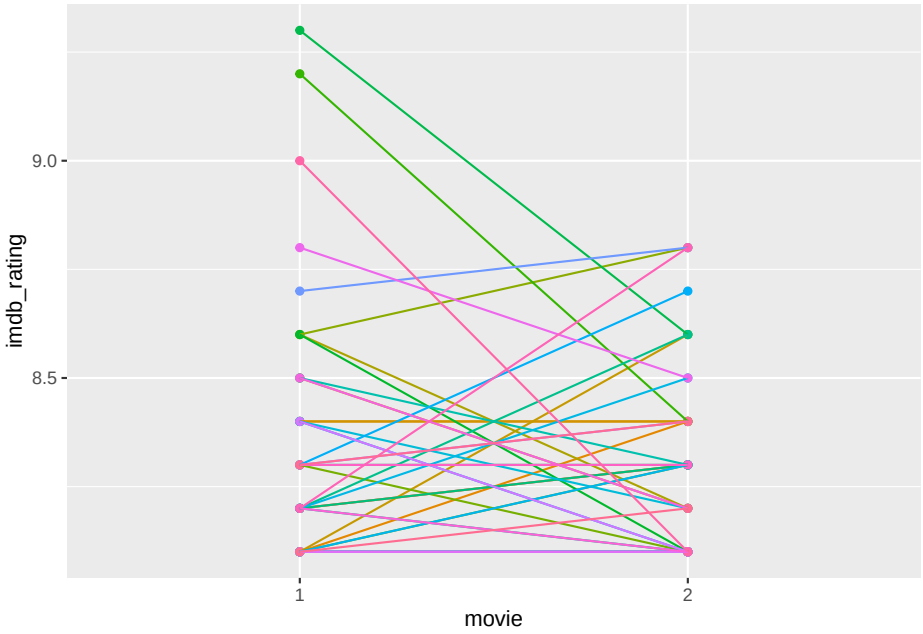
We want to know whether their earlier movies have been significantly more successful than others. We use the data frame `dir_mov` which only contains movies of directors who have two or more movies listen in the IMDb Top 250s.

```
dir_mov %>%
  ggplot(aes(x = movie, y = imdb_rating, fill = movie)) +
  geom_boxplot()
```



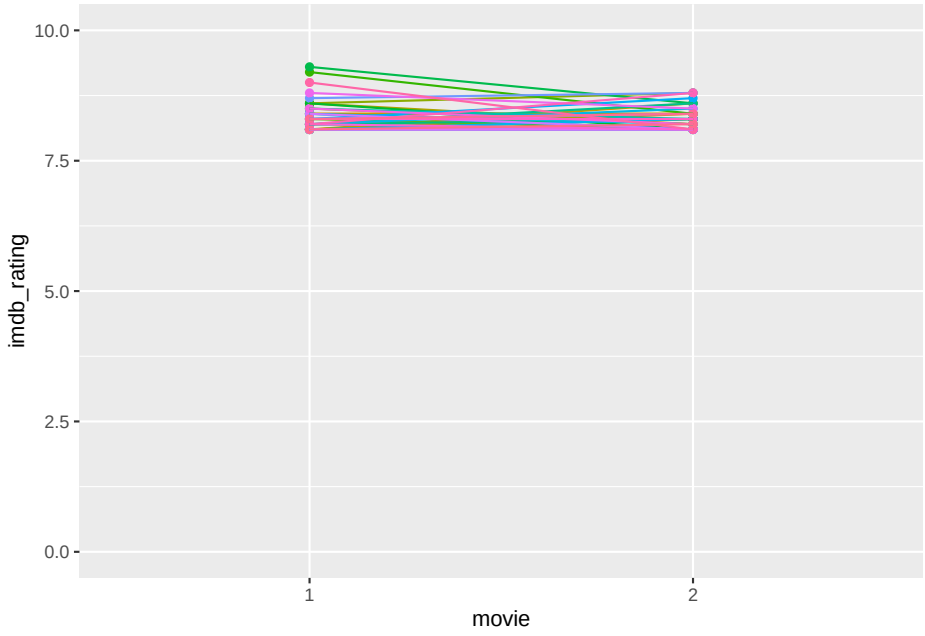
The boxplots look almost identical which suggests that the rating of movies in both groups has not changed significantly. However, the boxplot can only show a summary statistics for each group. Thus, it only implies that the movies in group `1` have about the same rating as the movies in `2`. If we want to visualise how the ratings have changed for each director from the first to the second movie, we can create a point plot and draw lines with `geom_line()` to connect the movies in each group. Line that movies up indicates that the second movie was rated higher than the first one and vice versa.

```
dir_mov %>%
  ggplot(aes(x = movie, y = imdb_rating, colour = director)) +
  geom_point() +
  geom_line(aes(group = director)) +
  # Remove the legend
  theme(legend.position = "none")
```



Based on this plot we have to revise our interpretation slightly. Directors who received particularly high ratings on their first movie (i.e. the top 3) scored much lower on the second movie. For once, we noticed from our boxplots that these movies count as outliers, and second, obtaining such high scores on a movie is tough to replicate. Needless to say, all these movies are rated as very good, otherwise they would not be in this list. It is worth noting that the way the y axis is scaled emphasises differences. Thus, a difference between a rating of 9 and 8.5 appears large. If we change the range of the y axis to '0-10', the differences appear marginal, but it reflects (1) the possible length of the scale (IMDb ratings range from 0-10) and (2) the magnitude in change relative to the entire scale.

```
dir_mov %>%
  ggplot(aes(x = movie, y = imdb_rating, colour = director)) +
  geom_point() +
  geom_line(aes(group = director)) +
  # Remove the legend
  theme(legend.position = "none")+
  # Manuall define the y axis range
  ylim(0,10)
```



Considering this last plot, we likely can predict what the statistical test will show. Table 10.2 summarises which tests and functions need to be performed if our data is parametric or non-parametric. In both cases, the functions are the same, but we need to add the attribute `paired = TRUE`. The interpretations between the unpaired and paired tests remain the same.

Table 10.2: Comparing two unpaired groups (effect size functions from package `effectsize`

Assumption	Test	Function for test	Effect size	Function for effect size
Parametric	T-Test	`t.test(paired = TRUE)`	Cohen's d	`cohens_d()`
Non-parametric	Wilcoxon Signed Rank Test	`wilcox.test(paired = TRUE)`	Rank biserial r	`rank_biserial()`

```
# PARAMETRIC COMPARISON
## Perform the comparison
(gcomp <- t.test(imdb_rating ~ movie, data = dir_mov, paired = TRUE))
##
##  Paired t-test
##
```

```
## data:  imdb_rating by movie
## t = 0.87171, df = 42, p-value = 0.3883
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.05504967  0.13877060
## sample estimates:
## mean of the differences
##              0.04186047

## Determine the effect size
(d <- cohens_d(imdb_rating ~ movie, data = dir_mov))
## Cohen's d |        95% CI
## -------------------------
## 0.16      | [-0.26, 0.58]
##
## - Estimated using pooled SD.

## Interpret the effect size
interpret_d(d$Cohens_d)
## [1] "very small"
## (Rules: cohen1988)

# NON-PARAMETRIC COMPARISON
## Perform the comparison
(gcomp <- wilcox.test(imdb_rating ~ movie, data = dir_mov, paired = TRUE))
##
##  Wilcoxon signed rank test with continuity correction
##
## data:  imdb_rating by movie
## V = 278, p-value = 0.5622
## alternative hypothesis: true location shift is not equal to 0

## Determine the effect size via rank biserial correlation
(d <- rank_biserial(imdb_rating ~ movie, data = dir_mov))
## r (rank biserial) |        95% CI
## --------------------------------
## 0.04              | [-0.20, 0.28]

## Interpret the effect size
interpret_rank_biserial(d$r_rank_biserial, rules = "cohen1988")
## [1] "very small"
## (Rules: cohen1988)
```

As expected, the paired tests reveal that the differences in rating between the first movie and the second movie are not significant and the effect sizes are

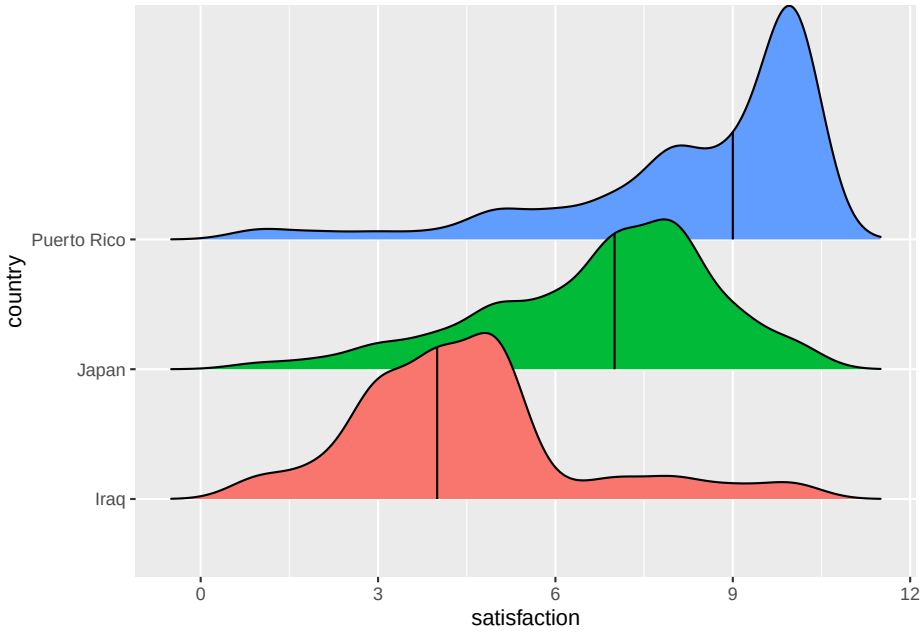
very small, irrespective of whether we treat our data as parametric or non-parametric.

10.2 Comparing more than two groups

10.2.1 Multiple unpaired groups

While we could plot boxplots again, let me share with you another approach, based on density plots. This has the added benefit that we can compare the distribution of data for each group and see whether the assumption of normality is likely met or not. On the other hand, we lose the option to identify any outliers.

```
wvs_nona %>%
  filter(country == "Iraq" | country == "Puerto Rico" | country == "Japan") %>%
  ggplot(aes(x = satisfaction, y = country, fill = country)) +
  ggridges::stat_density_ridges(bandwidth = 0.5,
                                quantile_lines = TRUE,
                                quantiles = (0.5)) +      # adds the median indicator
  # Remove legend
  theme(legend.position = "none")
```



anova()

kruskal.test()

10.2.2 Multiple paired groups

repeated measures ANOVA

Friedman Test

10.3 Comparing groups based on factors: The Chi-squared test

10.4 A cheatsheet to guide your own group comparisons

To summarise what we have covered in this section, we can consider

Chapter 11

Regression: Creating models to predict future observations

Chapter 12

Mixed-methods research: Analysing qualitative in R

Chapter 13

Where to go from here: The next steps in your R journey

13.1 GitHub: A Gateway to even more ingenious R packages

13.2 Books to read and expand your knowledge

13.3 Engage in regular online readings about R

- Tidyverse Blog
- R-blogger

13.4 Join the Twitter community and hone your skills

- #RStats
- TidyTuesday

Chapter 14

Exercises: Solutions

14.1 Solutions for 5.6

```
#1 ---------------
sqrt(25-16)+2*8-6
## [1] 13

#2 ---------------
"Five" == 5
## [1] FALSE

#3 ---------------
books <- list("Harry Potter and the Deathly Hallows",
              "The Alchemist",
              "The Davinci Code",
              "R For Dummies")

books
## [[1]]
## [1] "Harry Potter and the Deathly Hallows"
##
## [[2]]
## [1] "The Alchemist"
##
## [[3]]
## [1] "The Davinci Code"
##
## [[4]]
## [1] "R For Dummies"
```

```
# 4 ---------------
x_x <- function(number1, number2){       # Creates a function that takes 2 arguments
  result1 <- number1*number2             # Result1 multiplies the two arguments
  result2 <- sqrt(number1)               # Result2 computes the squareroot of the 1st
  result3 <- number1-number2             # Result3 subtracts the 2nd argument from the
  return(c(result1, result2, result3))   # Prints all three results into the console
}

x_x(2,3)
## [1]  6.000000  1.414214 -1.000000

# 5 ---------------

# First, install the package.

# Second, load the package with 'library()'
#         or use '::' to load only a particular function once.

# Third, call the function you are interested in.
```

Bibliography

Aguinis, H., Gottfredson, R. K., and Joo, H. (2013). Best-practice recommendations for defining, identifying, and handling outliers. *Organizational Research Methods*, 16(2):270–301.

Cambridge Dictionary (2021). Concatenate. Accessed: 2021-07-28.

Cohen, J. (1988). Statistical power analysis for the behavioral sciences new york. *NY: Academic*, page 54.

Cousineau, D. and Chartier, S. (2010). Outliers detection and treatment: a review. *International Journal of Psychological Research*, 3(1):58–67.

Dong, Y. and Peng, C.-Y. J. (2013). Principled missing data methods for researchers. *SpringerPlus*, 2(1):1–17.

Henson, R. K. (2001). Understanding internal consistency reliability estimates: A conceptual primer on coefficient alpha. *Measurement and evaluation in counseling and development*, 34(3):177–189.

Hu, L.-t. and Bentler, P. M. (1999). Cutoff criteria for fit indexes in covariance structure analysis: Conventional criteria versus new alternatives. *Structural equation modeling: a multidisciplinary journal*, 6(1):1–55.

Jakobsen, J. C., Gluud, C., Wetterslev, J., and Winkel, P. (2017). When and how should multiple imputation be used for handling missing data in randomised clinical trials–a practical guide with flowcharts. *BMC medical research methodology*, 17(1):1–10.

Leys, C., Ley, C., Klein, O., Bernard, P., and Licata, L. (2013). Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of experimental social psychology*, 49(4):764–766.

Little, R. J. (1988). A test of missing completely at random for multivariate data with missing values. *Journal of the American Statistical Association*, 83(404):1198–1202.

Müller, K. and Wickham, H. (2021). Column data types. Acccessd: 2021-08-05.

Nunally, J. C. (1967a). *Psychometric Theory*. New york: Mc Graw-Hill.

Nunally, J. C. (1967b). *Psychometric Theory (2nd edition)*. New york: Mc Graw-Hill.

Rubin, D. B. (1976). Inference and missing data. *Biometrika*, 63(3):581–592.

Schafer, J. L. (1999). Multiple imputation: a primer. *Statistical methods in medical research*, 8(1):3–15.

van Buuren, S. (2018). *Flexible imputation of missing data*. CRC press.

van Ginkel, J. R., Linting, M., Rippe, R. C., and van der Voort, A. (2020). Rebutting existing misconceptions about multiple imputation as a method for handling missing data. *Journal of personality assessment*, 102(3):297–308.

Wickham, H. (2021). The tidyverse style guide. Accessed: 2021-08-05.

Wickham, H. and Grolemund, G. (2016). *R for data science: import, tidy, transform, visualize, and model data*. " O'Reilly Media, Inc.".